

IF2211 Strategi Algoritma

IMPLEMENTASI ALGORITMA BFS DAN DFS DALAM

PENCARIAN SOLUSI PADA PERMASALAHAN TREASURE MAZE MAP

Laporan Tugas Besar II

Disusun untuk memenuhi tugas mata kuliah Strategi Algoritma
pada Semester II (dua) Tahun Akademik 2022/2023.



Oleh

Akbar Maulana Ridho	13521093
Nathania Callista	13521139
Alisha Listya Wardhani	13521171

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

DAFTAR ISI

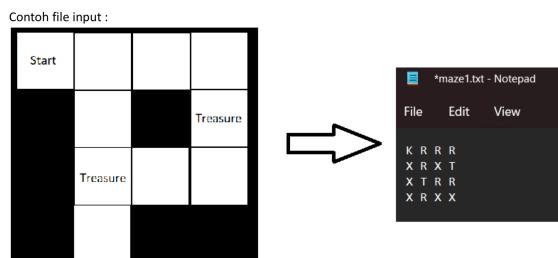
BAB I	
DESKRIPSI MASALAH	3
BAB II	
LANDASAN TEORI	4
2.1. Algoritma Pencarian	4
2.3. Aplikasi Pengembangan Aplikasi Desktop C#	6
BAB III	7
RANCANGAN PEMECAHAN MASALAH	7
2.1. Langkah Pemecahan Masalah	7
2.2. Pemetaan Algoritma Pencarian BFS dan DFS	9
2.3. Ilustrasi Penyelesaian dengan Rancangan	10
BAB IV	
ANALISIS PEMECAHAN MASALAH	11
3.1. Implementasi Program	11
3.2. Struktur Data Program	13
3.3. Pengujian Program	19
3.4. Analisis Perbandingan Algoritma	25
BAB V	
KESIMPULAN DAN SARAN	26
5.1. Kesimpulan	26
5.2. Saran	26
DAFTAR PUSTAKA	27
LAMPIRAN	28
Repository Github	28

BAB I

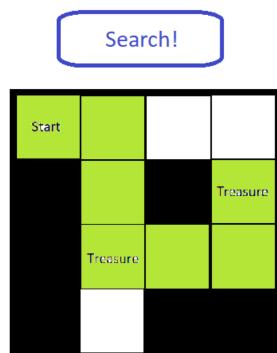
DESKRIPSI MASALAH

Labirin atau *maze* merupakan suatu permasalahan algoritma yang mengutamakan automasi metode untuk menyelesaikan masalah tersebut. Algoritma tersebut mencari jalan keluar dari labirin. Dalam Tugas Besar II Strategi Algoritma, diberikan sebuah permasalahan bertipe labirin yang berisi beberapa harta karun. Objektif utama permasalahan ini adalah mencari rute yang dapat menemukan seluruh harta karun. Dalam memecahkan permasalahan tersebut, digunakan algoritma Breadth First Search (BFS) dan Depth First Search (DFS). Hasil dan banyak langkah yang diperlukan dari kedua algoritma dapat berbeda. Prioritas arah simpul yang dibangkitkan dibebaskan (misalnya *left*, *right*, *up*, *down*). Pada permasalahan ini, tidak diperbolehkan pergerakan secara diagonal. Hasil rute tersebut kemudian divisualisasikan menggunakan GUI sederhana dengan bahasa C#.

Adapun dalam tugas ini juga dilakukan Travelling Salesman Problem (TSP) sehingga setelah didapatkan rute hasil, pemain dapat kembali ke titik awal. Dalam penyelesaian tugas ini, digunakan validitas penyelesaian, waktu eksekusi, dan kompleksitas algoritma sebagai tolak ukur keberhasilan algoritma. Berikut merupakan contoh masukan dan luaran program.



Gambar 1. Contoh Masukan Program



Gambar 2. Contoh Luaran Program

BAB II

LANDASAN TEORI

2.1. Algoritma Pencarian

2.2.1. Breadth First Search (BFS)

Algoritma Pencarian Breadth First Search merupakan algoritma traversal graf yang digunakan untuk menjelajahi simpul-simpul dalam struktur data grafik atau pohon. Pada BFS, algoritma dimulai dari simpul akar dan menjelajahi semua simpul di tingkat yang sama sebelum berpindah ke tingkat berikutnya. Hal ini berarti BFS mengeksplorasi semua node pada tingkat 1, semua node pada tingkat 2, dan seterusnya sampai semua node tereksplorasi.

Algoritma ini menggunakan struktur data *queue* untuk mencatat node yang perlu dieksplorasi di setiap level. Node dikunjungi sesuai urutan penambahannya ke dalam *queue*. Hal ini juga memastikan semua node telah tereksplorasi sebelum melanjutkan ke level berikutnya.

```
procedure BFS(input v:integer)
{ Traversal graf dengan algoritma pencarian BFS.
  Masukan : v adalah simpul awal kunjungan
  Keluaran : semua simpul yang dikunjungi dicetak ke layar}

Deklarasi
w: integer
q : antrian

procedure BuatAntrian(input/output q : antrian)
{ membuat antrian kosong, kepala(q) diisi nilai 0 }

procedure MasukAntrian(input/output q:antrian, input v: integer)
{ memasukkan v ke dalam antrian q pada posisi belakang }

procedure HapusAntrian(input/output q:antrian, input v:integer)
{ menghapus v dari kepala antrian q }

function AntrianKosong(inpit q:antrian) -> boolean
{ true jika antrian q kosong, false jika sebaliknya }

Algoritma
  BuatAntrian(q)
  write(v)
  dikunjungi(v) <- true
  MasukAntrian(q,v)
{ Kunjungi semua simpul graf selama antrian belum kosong }
  while not AntrianKosong(q) do
    HapusAntrian(q,v)
    for tiap simpul q yang bertetangga dengan v do
      if not dokunjungi(w) then
        write(w)
        MasukAntrian(q,w)
        dikunjungi(x) <- true
      endif
    endfor
  endwhile
{AntrianKosong(q)}
```

2.2.2. Depth First Search (DFS)

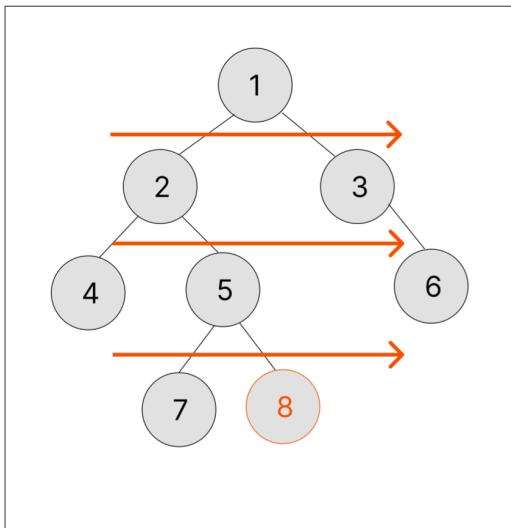
Sama seperti BFS, algoritma Pencarian Depth First Search merupakan algoritma traversal graf yang digunakan untuk menjelajahi simpul-simpul dalam struktur data grafik atau pohon. Bedanya adalah algoritma ini menjelajahi sebuah jalur sejauh mungkin sebelum *backtracking* ke jalur lain.

Algoritma ini menggunakan struktur data *stack* untuk melacak node yang perlu dieksplorasi.

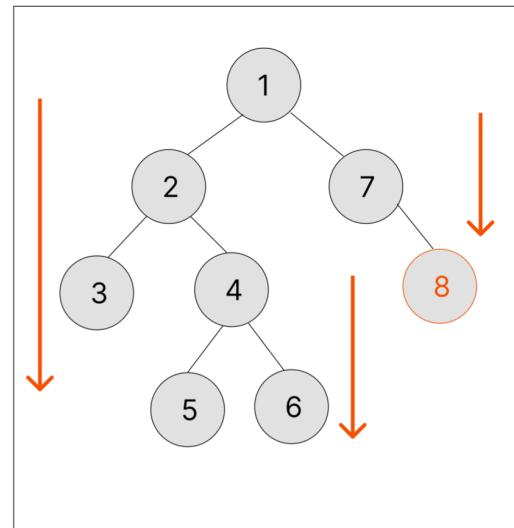
```
procedure DFS(input v:integer)
{ Mengunjungi seluruh simpul graf dengan algoritma pencarian DFS

Masukkan: v adalah simpul awal kunjungan
Keluaran: semua simpul yang dikunjungi ditulis ke layar }

Deklarasi
    w : integer
Algoritma
    write(v)
    dikunjungi[v] <- true
    for w <- 1 to n do
        if A[v,w]=1 then {simpul v dan simpul w bertetangga}
            if not dikunjungi[w] then
                DFS(w)
            endif
        endif
    endif
endif
```



Gambar 2.1.1. Skema Algoritma Pencarian BFS



Gambar 2.1.2. Skema Algoritma Pencarian DFS

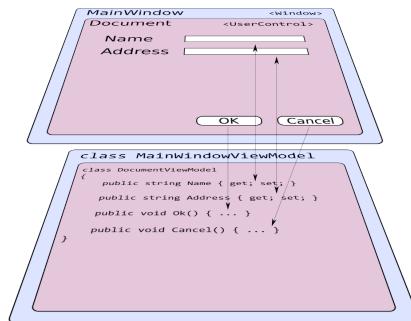
Pada Gambar 2.1.1. dan 2.1.2. memvisualisasikan perbedaan tahapan pengunjungan node pada setiap algoritma pencarian. BFS mengutamakan pencarian pada kedalaman yang sama. Sementara itu, DFS menuju kedalaman maksimal, kemudian melakukan *backtracking*.

2.3. Aplikasi Pengembangan Aplikasi Desktop C#

Pada Tugas Besar II Strategi Algoritma ini, program diimplementasikan dalam bahasa C#. Bahasa C# (disebut “*See Sharp*”) merupakan sebuah bahasa berorientasi objek yang berjalan di .NET. .NET merupakan sebuah sistem eksekusi berupa common language runtime (CLR) dan sebuah kumpulan kelas pustaka.

2.3.1. Avalonia UI

Untuk memvisualisasikan hasil, digunakan *framework* Avalonia UI. Avalonia merupakan framework cross-platform untuk DOTNET, menyediakan sistem styling yang fleksibel dan dapat dijalankan dalam berbagai sistem operasi, seperti Windows, Linux, dan MacOS. Avalonia menggunakan arsitektur pola MVVN (Model-View-ViewModel). Arsitektur ini menggunakan sistem *binding* avalonia untuk memisahkan logika aplikasi dan tampilan aplikasi.



Gambar 2.3.1.1. Ilustrasi Q=Window dan ViewModel

Bagian paling penting dalam pola MVVN adalah layer View dan layer ViewModel. View diimplementasikan dalam bentuk Windows dan UserControls, sementara ViewModels merupakan kelas .NET. Setiap View mempunyai ViewModel yang menampung semua logika View tersebut. Kedua hal ini dapat terhubung melalui *layer*. Karena layer ViewModel independen dari Avalonia, layer tersebut dapat dites secara unit seperti kode pada umumnya.

2.3.2. IDE

Dalam pembuatan tugas besar ini, IDE yang digunakan adalah Visual Studio dan JetBrains: Rider. Visual Studio merupakan IDE yang ideal untuk .NET dan C++. IDE tersebut menyediakan *error highlighting* dan compiler untuk .NET. Alasannya dipakai dua IDE dalam tugas ini adalah salah satu anggota kelompok terhalang sistem operasi (MacOS Mojave) sehingga Visual Studio versi terbaru tidak kompatibel. Alternatif untuk MacOS adalah menggunakan JetBrains: Rider.

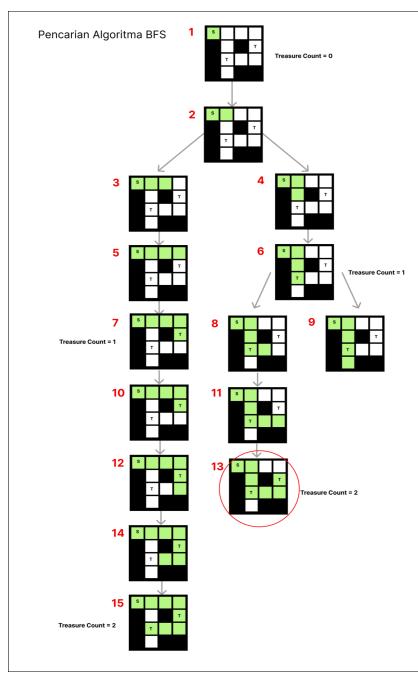
BAB III

RANCANGAN PEMECAHAN MASALAH

2.1. Langkah Pemecahan Masalah

Dalam pemecahan permasalahan tugas besar ini, penulis mengawali penggerjaan dengan analisis permasalahan untuk memudahkan penggerjaan. Analisis permasalahan tersebut menghasilkan beberapa bagian yang dijabarkan sebagai berikut:

1. Membuat GUI dalam bentuk desktop app yang dapat menerima *input* berupa peta pencarian, pilihan metode pencarian antara BFS atau DFS, *speed interval* yang menentukan kecepatan visualisasi pencarian, serta *toggle* TSP.
2. Membuat fungsi pencarian yang menerapkan algoritma BFS dan DFS. Penjelasan mengenai kedua algoritma akan dijelaskan pada poin di bawah. Pada algoritma, digunakan struktur data *stack* untuk DFS dan *queue* untuk BFS.
3. Membuat fungsi yang menangani visualisasi dari pencarian. Pembuatan graf dilakukan secara progresif dengan kecepatan yang ditentukan oleh pengguna. Dalam implementasinya, kami memanfaatkan Avalonia UI sebagai *tool* untuk menentukan *layout* dan hasil visualisasi dari graf.

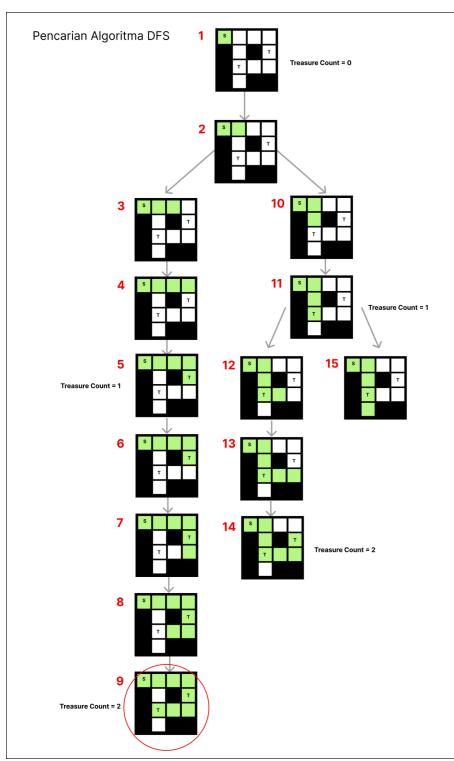


Berikut merupakan langkah penyelesaian dengan BFS:

1. Peta dimodelkan menjadi matriks.
2. Dimulai dari Start (KrustyKrab), lalu setiap simpul (koordinat) yang bertetanggaan dikunjungi. Kunjungi simpul yang terletak pada **kedalaman yang rendah terlebih dahulu**. Jika semua simpul pada kedalaman tersebut telah dikunjungi, baru dapat berpindah ke kedalaman berikutnya.
3. Pada setiap perpotongan, prioritaskan simpul yang belum pernah dikunjungi, kemudian kanan, bawah, kiri, kemudian atas.
5. Setiap simpul diperiksa apakah solusi sudah tercapai. Simpul yang telah dikunjungi sebelumnya disimpan dalam Route. Solusi tercapai jika Route mengandung semua harta karun.
6. Jika rute simpul mengandung semua harta karun, pencarian dapat selesai.

Gambar 2.1.1. Rancangan Algoritma BFS 7. Jika pengguna memilih menggunakan opsi TSP, maka pencarian dimulai dari simpul terakhir sampai mendapatkan KrustyKrab sebagai simpul akhir yang baru.

Pada Gambar 2.1.1., terlihat bahwa pengecekan memprioritaskan arah kanan terlebih dahulu dibandingkan arah bawah (diilustrasikan pada pemilihan langkah 3 dan 4). Perhatikan bahwa langkah penyelesaian berakhir pada langkah ke-13 karena semua harta karun sudah diperoleh pada langkah tersebut. Pada algoritma BFS, pengecekan simpul tidak beruntun membentuk sebuah rute. Hal ini menyebabkan visualisasi pengecekan dapat terlihat acak dan terkesan melompat-lompat.



Gambar 2.1.2. Rancangan Algoritma DFS

Berikut merupakan langkah penyelesaian dengan DFS:

1. Peta dimodelkan menjadi matriks.
2. Dimulai dari Start (KrustyKrab), lalu setiap simpul (koordinat) yang bertetanggaan dikunjungi. Kunjungi simpul dengan memprioritaskan simpul yang belum pernah dikunjungi, kemudian kanan, bawah, kiri, kemudian atas.
3. Setiap simpul diperiksa apakah solusi sudah tercapai. Simpul yang telah dikunjungi sebelumnya disimpan dalam Route. Solusi tercapai jika Route mengandung semua harta karun.
4. Ketika sudah tidak ada simpul lagi yang bisa dikunjungi dan jumlah harta karun masih kurang, dilakukan *backtrack*.
5. Agenda pengunjungan direpresentasikan sebagai sebuah **stack (LIFO)**.
6. Jika rute simpul mengandung semua harta karun, pencarian dapat selesai.
7. Jika pengguna memilih menggunakan opsi TSP, maka pencarian dimulai dari simpul terakhir sampai mendapatkan KrustyKrab sebagai simpul akhir yang baru.

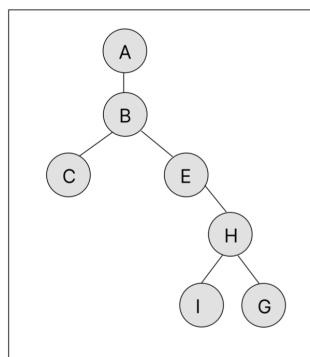
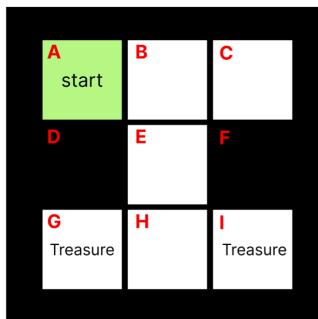
Perhatikan pada Gambar 2.1.2, bahwa hasil penyelesaian berbeda dengan algoritma BFS karena langkah ke-9 adalah langkah pertama ditemukannya semua harta karun.

2.2. Pemetaan Algoritma Pencarian BFS dan DFS

Program akan menerima masukan pengguna berupa peta yang berisi jalur, letak harta karun, letak mulai (KrustyKrab). Dari letak KrustyKrab, akan dihasilkan graf yang menggambarkan proses penelusuran yang terjadi. Pendekatan graf yang dilakukan pada tugas besar ini merupakan graf dinamis. Graf terbentuk dalam proses pencarian, simpul direpresentasikan dengan struktur data buatan route. Route menampung informasi berikut:

1. Koordinat yang sedang dikunjungi
2. Koordinat-koordinat sebelumnya
3. Koordinat treasure yang telah diambil
4. Jumlah treasure yang telah diambil

Pada setiap koordinat yang sedang dikunjungi, akan dicek tetangganya apakah bisa dikunjungi atau tidak. Jika bisa, rute menuju tetangga tersebut akan dimasukkan ke queue/stack. Berikut merupakan visualisasi penyelesaian masalah.



Gambar 2.2.1. Skema Permasalahan

Simpul-e	Simpul Hidup
A	B _A
B _A	C _{AB} E _{AB}
C _{AB}	E _{AB} B _{ABC}
E _{AB}	B _{ABC} H _{ABE}
B _{ABC}	H _{ABE} C _{ABC} E _{ACB}
H _{ABE}	E _{ACB} I _{ABEH} G _{ABEH}
E _{ACB}	I _{ABEH} G _{ABEH} H _{ABCBE}
I _{ABEH}	G _{ABEH} H _{ABCBE} H _{ABEHI}
G _{ABEH}	H _{ABCBE} H _{ABEHI} H _{ABEHG}
H _{ABCBE}	H _{ABEHI} H _{ABEHG}
H _{ABEHI}	H _{ABEHG} G _{ABEHIH}
H _{ABEHG}	G _{ABEHIH} I _{ABEHGH}
G _{ABEHIH}	

Gambar 2.2.2. Queue BFS

Simpul-e	Simpul Hidup
A	B _A
B _A	C _{AB} E _{AB}
C _{AB}	B _{ABC} E _{AB}
B _{ABC}	E _{ACB} E _{AB}
E _{ACB}	H _{ABCBE} E _{AB}
H _{ABCBE}	I _{ABCEH} G _{ABCBEH} E _{AB}
I _{ABCEH}	H _{ABCBEH} G _{ABCBEH} E _{AB}
H _{ABCBEH}	G _{ABCBEHH} G _{ABCBEH} E _{AB}
G _{ABCBEHH}	

Gambar 2.2.3. Stack DFS

Prioritas pencarian: simpul yang belum dikunjungi, kanan, bawah, kiri, kemudian atas. Pencarian akan terus dilakukan sampai semua simpul habis dikunjungi atau semua *treasure* sudah terambil. Hal ini memungkinkan adanya *backtracking*.

2.3. Ilustrasi Penyelesaian dengan Rancangan

Tabel 2.3.1. Ilustrasi Penyelesaian Persoalan Lain

Permasalahan	BFS	DFS																								
	<p>Pencarian Algoritma BFS</p> <table border="1"> <thead> <tr> <th>Simpul-e</th><th>Simpul Hidup</th></tr> </thead> <tbody> <tr> <td>B</td><td>C_B E_B A_B</td></tr> <tr> <td>C_B</td><td>E_B A_B</td></tr> <tr> <td>E_B</td><td>A_B H_{BE}</td></tr> <tr> <td>A_B</td><td>H_{BE} D_{BA}</td></tr> <tr> <td>H_{BE}</td><td></td></tr> </tbody> </table>	Simpul-e	Simpul Hidup	B	C _B E _B A _B	C _B	E _B A _B	E _B	A _B H _{BE}	A _B	H _{BE} D _{BA}	H _{BE}		<p>Pencarian Algoritma DFS</p> <table border="1"> <thead> <tr> <th>Simpul-e</th><th>Simpul Hidup</th></tr> </thead> <tbody> <tr> <td>B</td><td>C_B E_B A_B</td></tr> <tr> <td>C_B</td><td>E_B A_B</td></tr> <tr> <td>E_B</td><td>H_{BE} A_B</td></tr> <tr> <td>A_B</td><td></td></tr> <tr> <td>H_{BE}</td><td></td></tr> </tbody> </table>	Simpul-e	Simpul Hidup	B	C _B E _B A _B	C _B	E _B A _B	E _B	H _{BE} A _B	A _B		H _{BE}	
Simpul-e	Simpul Hidup																									
B	C _B E _B A _B																									
C _B	E _B A _B																									
E _B	A _B H _{BE}																									
A _B	H _{BE} D _{BA}																									
H _{BE}																										
Simpul-e	Simpul Hidup																									
B	C _B E _B A _B																									
C _B	E _B A _B																									
E _B	H _{BE} A _B																									
A _B																										
H _{BE}																										
	<p>Pencarian Algoritma BFS</p>	<p>Pencarian Algoritma DFS</p>																								
	Queue dan Stack tidak ditulis karena terlalu panjang.																									

BAB IV

ANALISIS PEMECAHAN MASALAH

3.1. Implementasi Program

Berikut merupakan pseudocode algoritma utama yang memuat logika utama pada program.

```
{ Algoritma Pengecekan Moveable Unit }
{ Algoritma General Untuk BFS dan DFS }

function Moveable(input: Coordinate c, output: boolean) {
    if (c.X < 0 OR c.X >= mazeMap.size OR c.Y < 0 OR c.Y >= mazeMap.size) then
        -> false
    if (mazeMap.GetElement(c) = Element.Tunnel || mazeMap.GetElement
        = Element.Treasure || mazeMap.GetElement = Element.KrustyKrab) then
        -> true
    else then
        -> false

function AvailableMovement(input: route, output: List) {
    top <- route.currCoordinate.Top()
    if (movable(top)) then
        isTreasure = mazeMap.getElement(top) = Element.Treasure
        newRoute = new Route(isTreasure, top, route)
        Tuple <int, Movement, Route> t = new(route.getVisited(top), UP, route)
        list.Add(t)

    left <- route.currCoordinate.Left()
    if (movable(left)) then
        isTreasure = mazeMap.getElement(left) = Element.Treasure
        newRoute = new Route(isTreasure, left, route)
        Tuple <int, Movement, Route> l = new(route.getVisited(left), LEFT, route)
        list.Add(l)

    bottom <- route.currCoordinate.Bottom()
    if (movable(bottom)) then
        isTreasure = mazeMap.getElement(bottom) = Element.Treasure
        newRoute = new Route(isTreasure, bottom, route)
        Tuple <int, Movement, Route> b = new(route.getVisited(bottom), DOWN, route)
        list.Add(b)

    right <- route.currCoordinate.Right()
    if (movable(right)) then
        isTreasure = mazeMap.getElement(right) = Element.Treasure
        newRoute = new Route(isTreasure, right, route)
        Tuple <int, Movement, Route> r = new(route.getVisited(right), RIGHT, route)
        list.Add(r)

{ Algoritma BFS }

procedure AddAvailableMovement(Route route) {
    list <- AvailableMovement(route)
    sorted <- list.OrderBy(visitedCount).ThenBy(MovementPriorities)
    minVisited = sorted.Min(visitedCount)
    for (e in sorted) do
        if (prevCoordinate.Count = 0 OR sorted.Count = 1
        OR CurrentCoordinate != route.prevCoordinate AND visitedCount = minVisited) then
            enqueue(e.currCoordinate)
        else if (e.CurrCoordinate = route.prevCoordinate) then
            if (sorted.Count > 1 AND minVisited != sorted[1].visitedCount
                AND e.VisitedCount = minVisited)
                enqueue(e.CurrCoordinate)}
```

```
procedure Visit() {
    currRoute <- available.Dequeue
    currRoute.setVisitedRoute(currRoute.currCoordinate)
    result.Move(currRoute)
    AddAvailableMovement(currRoute)

procedure Solve(){
    int i <- 0
    do {
        Visit()
        i <- i+1
    } while (available.Count > 0 AND currRoute.TreasureCount < mazeMap.TotalTreasure)

    if (currRoute.TreasureCount = mazeMap.TotalTreasure) then
        if (TSP) then
            available.clear()
            AddAvailableMovement(currRoute)
            do {
                Visit()
                i <- i+1
            } while (available.Count > 0
                    AND mazeMap.getElement(currRoute.currCoordinate != KrustyKrab))
        result.setSolved()
        routes <- currRoute.prevCoordinates
        routes.Add(currCoordinate)
    }
}

{ Algoritma DFS }

procedure AddAvailableMovement(Route route) {
    list <- AvailableMovement(route)
    sorted <- list.OrderBy(visitedCount).ThenBy(MovementPriorities)

    for (e in sorted) do
        temp <- currRoute.prevCoordinate()
        if (e.Route.CurrCoordinate = route.prevCoordinate() AND sorted.Count = 1) then
            continue
        available.Push(e.Route)

procedure Visit() {
    currRoute <- available.Pop
    setVisited(currRoute.currCoordinate)
    result.Move(currRoute)
    AddAvailableMovement(currRoute)

procedure Solve(){
    do {
        Visit()
    } while (available.Count > 0 AND currRoute.TreasureCount < mazeMap.TotalTreasure)

    if (currRoute.TreasureCount = mazeMap.TotalTreasure) then
        if (TSP) then
            do {
                Visit()
            } while (available.Count > 0
                    AND mazeMap.getElement(currRoute.currCoordinate != KrustyKrab))
        result.setSolved()
    }
}
```

3.2. Struktur Data Program

Pada tugas besar ini, terdapat beberapa struktur data yang digunakan, yaitu:

1. **Stack**, Stack merupakan struktur data yang menggunakan prinsip LIFO (*Last In First Out*) yang penggunaannya dengan memanfaatkan metode Push (memasukkan elemen) dan Pop (mengeluarkan elemen). Struktur data ini diimplementasikan pada algoritma DFS. Penggunaan stack pada program divisualisasikan pada gambar 2.2.3.
2. **Queue**, Queue merupakan struktur data yang menggunakan prinsip FIFO (*First In First Out*) yang penggunaannya memanfaatkan metode Enqueue (memasukkan elemen) dan Dequeue (mengeluarkan elemen).
3. **List**, List merupakan struktur data yang menyimpan elemen atau objek dalam bentuk list. Struktur data ini digunakan untuk menyimpan koordinat-koordinat yang dapat dikunjungi.
4. **Tuple**, Tuple merupakan struktur data yang menyimpan berbagai jenis elemen dalam satu variabel. Pada program ini, tuple digunakan untuk menyimpan *state* simpul yang dikunjungi, meliputi berapa kali telah dikunjungi, letak simpul, dan apakah mengandung harta karun.
5. **HashSet**, HashSet merupakan struktur data yang menyimpan elemen unik (berperilaku seperti himpunan). Struktur data ini digunakan untuk pengecekan jumlah *treasure* dalam program.
6. **Route** (struktur data buatan), yang mengandung simpul-simpul yang telah dikunjungi sebelumnya, state simpul tersebut, dan simpul yang sedang dikunjungi.
7. **Coordinate** (struktur data buatan).

Dalam pengembangan program ini, dibuat kelas-kelas yang mengimplementasikan struktur data tersebut. Berikut merupakan penjelasan dari setiap kelas yang dibuat.

Tabel 3.2.1. Kelas Coordinate

Nama kelas: Coordinate	
Atribut	
Method	
Tuple<int, int> Coordinate	Tipe Coordinate untuk menyimpan letak pada peta
public Coordinate Top()	Mengembalikan letak Coordinate di atas Coordinate.
public Coordinate Bottom()	Mengembalikan letak Coordinate di bawah Coordinate.
public Coordinate Left()	Mengembalikan letak Coordinate di kiri Coordinate.
public Coordinate Right()	Mengembalikan letak Coordinate di kanan Coordinate.
public static bool operator==(Coordinate c)	Mengembalikan true jika Coordinat c sama dengan Coordinate.
public static bool operator!=(Coordinate c)	Mengembalikan true jika Coordinat c tidak sama dengan Coordinate.

Tabel 3.2.2. Kelas Route

Nama kelas: Route	
Atribut	
private int treasureCount	Tipe integer yang menyimpan jumlah treasure dalam rute tersebut
protected bool isTreasure	Tipe boolean yang menyimpan apakah koordinat tersebut mengandung treasure
protected HashSet<Coordinate> treasures	Tipe HashSet yang menyimpan koordinat treasure yang telah dikunjungi
protected Coordinate currCoordinate	Tipe Coordinate yang menyimpan tempat simpul telah dikunjungi
protected List <Tuple<bool,Coordinate>> prevCoordinate	Tipe List yang menyimpan koordinat sebelumnya yang telah dikunjungi
Method	
public bool IsTreasure	Getter boolean apakah simpul mengandung treasure
public Coordinate CurrentCoordinate	Getter koordinat simpul yang sedang dikunjungi
public int TreasureCount	Getter jumlah treasure dalam rute
public Coordinate Prevcoordinate()	Getter simpul yang telah dikunjungi sebelumnya
public List <Tuple<bool,Coordinate>> prevCoordinates	Getter list simpul-simpul yang telah dikunjungi sebelumnya
public Route(bool isTreasure, Coordinate c)	Konstruktor Route.
public Route(bool isTreasure, Coordinate c, Route prev)	Konstruktor Route.

Tabel 3.2.3. Kelas MazeMap

Nama kelas: MazeMap	
Atribut	
protected List<List<Element>> map	Tipe Matriks yang berisi elemen dari peta
protected int totalTreasure	Tipe integer yang menyimpan banyak treasure pada peta
public Coordinate StartPosition	Tipe koordinat yang menyimpan Koordinat awal
public int size	Tipe integer yang menyimpan ukuran matriks
Method	
public MazeMap(string filePath)	Konstruktor MazeMap dengan membaca file txt
public Element GetElement(Coordinate c)	Mengembalikan tipe elemen dari koordinat

Tabel 3.2.4. Kelas BFS

Nama kelas: BFS
Atribut

<code>protected MazeMap mazeMap</code>	Tipe MazeMap yang menyimpan peta
<code>protected BFSResult result</code>	Tipe BFSResult yang menyimpan hasil
<code>protected bool TSP</code>	Tipe boolean yang menyimpan apakah solusi TSP
<code>private Queue<Route> available</code>	Tipe Queue yang menyimpan route yang akan dikunjungi
<code>protected Route currRoute</code>	Tipe Route yang menyimpan route yang sedang dikunjungi
Method	
<code>public BFS(MazeMap m, bool TSP)</code>	Konstruktor BFS
<code>public BFSResult GetResult()</code>	Metode untuk mengembalikan hasil
<code>protected bool Moveable(Coordinate c)</code>	Metode untuk mengetahui apakah koordinat tersebut mungkin dikunjungi
<code>protected List<Tuple<int, Movement, RouteBFS>> AvailableMovement(RouteBFS route)</code>	Metode untuk mengetahui arah yang mungkin untuk bergerak
<code>protected void AddAvailableMovement(RouteBFS route)</code>	Metode untuk menambahkan rute ke dalam queue
<code>protected void Visit()</code>	Metode untuk mengunjungi sebuah simpul
<code>public void Solve()</code>	Metode untuk melakukan penyelesaian

Tabel 3.2.5. Kelas DFS

Nama kelas: DFS	
Atribut	
<code>protected MazeMap mazeMap</code>	Tipe MazeMap yang menyimpan peta
<code>protected BFSResult result</code>	Tipe BFSResult yang menyimpan hasil
<code>private List<List<int>> visitedCount</code>	Tipe List yang menyimpan
<code>protected bool TSP</code>	Tipe boolean yang menyimpan apakah solusi TSP
<code>private Stack<Route> available</code>	Tipe Stack yang menyimpan route yang akan dikunjungi
<code>protected Route currRoute</code>	Tipe Route yang menyimpan route yang sedang dikunjungi
Method	
<code>public DFS(MazeMap m, bool TSP)</code>	Konstruktor DFS
<code>public DFSResult GetResult()</code>	Metode untuk mengembalikan hasil
<code>protected bool Moveable(Coordinate c)</code>	Metode untuk mengetahui apakah koordinat tersebut mungkin dikunjungi
<code>protected List<Tuple<int, Movement, RouteBFS>> AvailableMovement(RouteBFS route)</code>	Metode untuk mengetahui arah yang mungkin untuk bergerak
<code>protected void AddAvailableMovement(RouteBFS route)</code>	Metode untuk menambahkan rute ke dalam queue
<code>protected void Visit()</code>	Metode untuk mengunjungi sebuah simpul

public void Solve()	Metode untuk melakukan penyelesaian
---------------------	-------------------------------------

Kelas lainnya merupakan kelas interface yang menghubungkan antarmuka dengan algoritma. Kelas tersebut diantaranya: BFSResult, DFSResult, dan IResult.

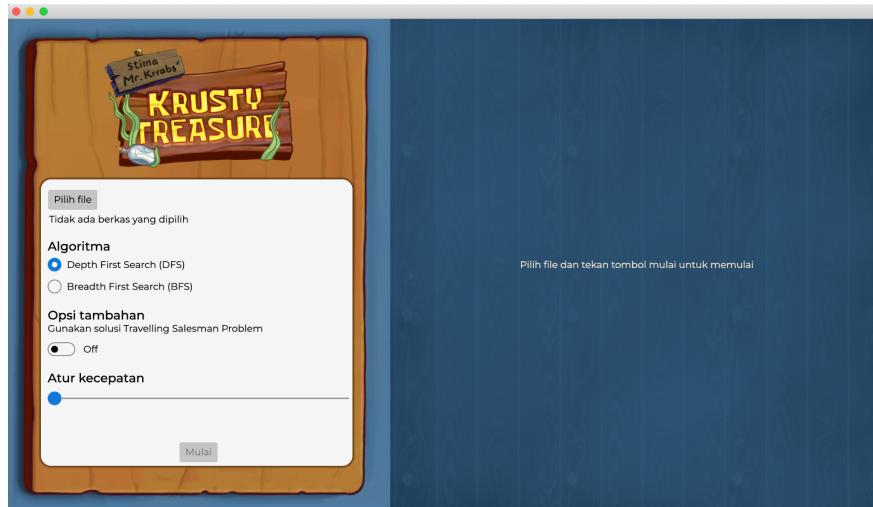
Sementara itu, dalam pembuatan Graphical User Interface (GUI), terdiri dari file-file berikut. Views menangani tampilan dan *styling* pada GUI, sementara ViewModels menjadi referensi *binding variable* yang terdapat pada tampilan program.

Tabel 3.2.6. File pendukung ViewModels dan Views

ViewModels	Views	Deskripsi
FormViewModel.cs	FormView.axaml FormView.axaml.cs	Tampilan yang menangani masukan pengguna pada awal program.
GridDirtViewModel.cs	GridDirtView.axaml GridDirtView.axaml.cs	Menangani tampilan unit Dirt pada peta
GridKrustyViewModel.cs	GridKrustyView.axaml GridKrustyView.axaml.cs	Menangani tampilan unit KrustyKrab pada peta
GridTreasureViewModel.cs	GridTreasureView.axaml GridTreasureView.axaml.cs	Menangani tampilan unit Treasure pada peta
GridTunnelViewModel.cs	GridTunnelView.axaml GridTunnelView.axaml.cs	Menangani tampilan unit Tunnel pada peta
GridViewModelBase.cs	-	Kelas dasar untuk menangani tampilan grid
MainWindowViewModel.cs	MainWindow.axaml MainWindow.axaml.cs	Window utama program. Terdiri atas dua bagian. Bagian kiri bisa berisi FormView dan Result View. Bagian kanan bisa berisi MazeEmpty dan MazeView
MazeEmptyViewModel.cs	MazeEmptyView.axaml MazeEmptyView.axaml.cs	Menangani tampilan program (peta kosong) ketika belum menerima input
MazeViewModel.cs	MazeView.axaml MazeView.axaml.cs	Menangani tampilan peta ketika sudah menerima hasil pencarian rute
ResultViewModel.cs	ResultView.axaml ResultView.axaml.cs	Menangani tampilan hasil penyelesaian, seperti banyaknya langkah, arah rute, dan waktu eksekusi
ViewModelBase.cs	-	Basis Model

3.3. Tata Cara Penggunaan Program

Berikut merupakan bentuk program ketika baru dibuka. Fitur yang tersedia adalah sebagai berikut:



Gambar 3.3.1. Tampilan Utama Program

Fitur-fitur yang terdapat pada program yaitu sebagai berikut:

1. *Button “Pilih File”* yang mengarahkan pengguna untuk memilih file konfigurasi peta.
2. *Textbox* berisi nama file yang dipilih. Jika file tidak ditemukan atau format file peta salah, maka akan menampilkan pesan kesalahan.
3. *Radio Button* pemilihan algoritma antara BFS ataupun DFS
4. *Toggle Button* untuk memilih apakah ingin solusi TSP (Travelling Salesman Problem)
5. *Slider* untuk mengatur *interval* kecepatan pengguna



Gambar 3.3.2. Tampilan Penyelesaian Program

Berikut merupakan tampilan akhir hasil penyelesaian, memuat rute jumlah langkah, jumlah simpul diperiksa serta waktu eksekusi.

KrustyKrab (Titik mulai)	Pemain DFS	Pemain BFS	Tunnel (dapat dilewati)	Dirt (tidak dapat dilewati)
Logo	Harta karun (sudah terbuka)	Harta karun (belum terbuka)	Tunnel (sudah dilewati)	Rute penyelesaian

Gambar 3.3.3. Legenda Peta

Berikut merupakan legenda dari setiap icon pada peta. Jalur pada peta terdapat 4: Tunnel, Tunnel yang sudah dilewati, Dirt, dan rute penyelesaian. Hal ini berfungsi untuk memudahkan dalam visualisasi. Pada visualisasi DFS, terlihat bahwa pemain menyusuri peta secara struktural.

Pada visualisasi BFS, terlihat bahwa pemain menyusuri peta secara lompat-lompat. Walau terlihat tidak runut, hal ini adalah hal yang disengaja. Dengan Tunnel berwarna *orange* menandakan rute mana pemain tersebut sedang berada. Alasan dipilihnya jenis visualisasi ini adalah penulis merasa bahwa visualisasi ini lebih benar secara teoritis dibandingkan jika melakukan visualisasi backtrack (kembali ke simpul sebelumnya).

3.3. Pengujian Program

Berikut merupakan pengujian program menggunakan penyelesaian BFS dan DFS.

Tabel 3.3.1. Pengujian Program Tanpa TSP

Permasalahan Tanpa TSP	Penyelesaian						
Permasalahan 1	<p>Penyelesaian DFS</p>  <p>Hasil Pencarian</p> <p>Algoritma Depth First Search (DFS) tanpa TSP</p> <table border="1"> <thead> <tr> <th>Langkah</th> <th>Simpul diperiksa</th> <th>Waktu Eksekusi</th> </tr> </thead> <tbody> <tr> <td>113</td> <td>114</td> <td>2,00 ms</td> </tr> </tbody> </table> <p>Rute</p> <pre>RIGHT - RIGHT - RIGHT - UP - RIGHT - UP - UP - LEFT - RIGHT - DOWN - DOWN - DOWN - LEFT - LEFT - UP - LEFT - UP - LEFT - LEFT - LEFT - LEFT - LEFT - DOWN - DOWN - RIGHT - LEFT - LEFT - DOWN - DOWN - RIGHT - RIGHT - RIGHT - DOWN - DOWN - RIGHT - RIGHT - RIGHT - RIGHT - DOWN - LEFT - DOWN - DOWN - RIGHT - LEFT - LEFT - LEFT - LEFT -</pre> <p>Kembali</p>	Langkah	Simpul diperiksa	Waktu Eksekusi	113	114	2,00 ms
Langkah	Simpul diperiksa	Waktu Eksekusi					
113	114	2,00 ms					
	<p>Analisis: Penyelesaian ditemukan. Simpul yang diperiksa tergolong sedikit, tetapi terlihat bahwa penyelesaian tidak optimal.</p>						
	<p>Penyelesaian BFS</p>  <p>Hasil Pencarian</p> <p>Algoritma Breath First Search (BFS) tanpa TSP</p> <table border="1"> <thead> <tr> <th>Langkah</th> <th>Simpul diperiksa</th> <th>Waktu Eksekusi</th> </tr> </thead> <tbody> <tr> <td>46</td> <td>45918</td> <td>1,08 s</td> </tr> </tbody> </table> <p>Rute</p> <pre>RIGHT - RIGHT - UP - UP - LEFT - UP - UP - LEFT - LEFT - LEFT - LEFT - LEFT - DOWN - DOWN - DOWN - DOWN - RIGHT - RIGHT - RIGHT - RIGHT - RIGHT - UP - DOWN - DOWN - DOWN - DOWN - LEFT - LEFT - LEFT - LEFT - DOWN - DOWN - RIGHT - RIGHT - RIGHT - RIGHT - RIGHT - RIGHT - RIGHT - RIGHT - RIGHT - RIGHT - UP - UP - UP -</pre> <p>Kembali</p>	Langkah	Simpul diperiksa	Waktu Eksekusi	46	45918	1,08 s
Langkah	Simpul diperiksa	Waktu Eksekusi					
46	45918	1,08 s					
	<p>Analisis: Penyelesaian ditemukan. Terjadi pembengakkan terhadap jumlah simpul yang diperiksa. Tetapi perlu diperhatikan bahwa solusi yang didapatkan merupakan solusi optimal. Waktu eksekusi juga menunjukkan kenaikan yang sangat signifikan.</p>						

Permasalahan 2	<p>File tidak dikenali atau salah format file map salah.</p>						
Permasalahan 3	<p>Penyelesaian DFS</p> <p>Hasil Pencarian</p> <p>Algoritma Depth First Search (DFS) tanpa TSP</p> <table border="1"> <thead> <tr> <th>Langkah</th> <th>Simpul diperiksa</th> <th>Waktu Eksekusi</th> </tr> </thead> <tbody> <tr> <td>7</td> <td>8</td> <td>0,00 ms</td> </tr> </tbody> </table> <p>Rute RIGHT - RIGHT - RIGHT - DOWN - DOWN - LEFT - LEFT</p>	Langkah	Simpul diperiksa	Waktu Eksekusi	7	8	0,00 ms
Langkah	Simpul diperiksa	Waktu Eksekusi					
7	8	0,00 ms					
	<p>Analisis: Penyelesaian ditemukan. Simpul yang diperiksa normal, tetapi langkah penyelesaian belum optimal.</p>						
	<p>Penyelesaian BFS</p> <p>Hasil Pencarian</p> <p>Algoritma Breadth First Search (BFS) tanpa TSP</p> <table border="1"> <thead> <tr> <th>Langkah</th> <th>Simpul diperiksa</th> <th>Waktu Eksekusi</th> </tr> </thead> <tbody> <tr> <td>6</td> <td>14</td> <td>1,00 ms</td> </tr> </tbody> </table> <p>Rute RIGHT - DOWN - DOWN - RIGHT - RIGHT - UP</p> <p>Analisis: Penyelesaian ditemukan. Simpul yang diperiksa hampir 2x lipat DFS, tetapi penyelesaian lebih efisien.</p>	Langkah	Simpul diperiksa	Waktu Eksekusi	6	14	1,00 ms
Langkah	Simpul diperiksa	Waktu Eksekusi					
6	14	1,00 ms					

Permasalahan 4	Penyelesaian DFS						
XRRRRRRRRRRRTXRRRTX TRXRRRRRRRRRRTXXRX XRXRTTXXTRRRRRRXRX XRXXXTTTTTXXXRRRX TRXTRXXXRXXXRRRXRX XRXRTXRRXTTXXRXRTT RRRRRRRRRRRRRTTXXT XRXXRRXTXXXXTTTXXX TRRXXXRXXRRRXRRRRRT XRTTRRXXTXXRRXXXXXR RXTTRXRXTKTRXRRRRRT RRXXRXRXRTXRXRTXXXXT XRXXRXXXRXXRRRRRTT RRRTTRXXRXRXXXXXXXT TXXXTXXRXRXRRRRRRRTT XXRXXXRXRRXRRTXRX RXRXTXRXRXRTXRXRXR TXRXRXRXRXRTXXRXRXR RXRXRXRXRXRTXRXRTXRX RRRRRRRRTRRRRRRRRRR	 <p>Hasil Pencarian</p> <p>Algoritma Depth First Search (DFS) tanpa TSP</p> <table border="1"> <thead> <tr> <th>Langkah</th> <th>Simpul diperiksa</th> <th>Waktu Eksekusi</th> </tr> </thead> <tbody> <tr> <td>898</td> <td>899</td> <td>59.00 ms</td> </tr> </tbody> </table> <p>Kembali</p> 	Langkah	Simpul diperiksa	Waktu Eksekusi	898	899	59.00 ms
Langkah	Simpul diperiksa	Waktu Eksekusi					
898	899	59.00 ms					
	<p>Analisis: Penyelesaian ditemukan. Namun, solusi belum merupakan solusi yang efisien. Simpul yang diperiksa linier dengan jumlah langkah.</p> <p>Penyelesaian BFS</p> 						

Berikut merupakan penyelesaian masalah dengan TSP sebagai bonus dari spesifikasi...

Tabel 3.3.2. Pengujian Program Dengan TSP

Permasalahan 6	Penyelesaian DFS						
 <pre>XRRXR TXXRX RRKRT RXTXX TRRRX</pre>	<p>Penyelesaian DFS</p>  <p>Hasil Pencarian</p> <p>Algoritma Depth First Search (DFS) dengan TSP</p> <table border="1"> <thead> <tr> <th>Langkah</th> <th>Simpul diperiksa</th> <th>Waktu Eksekusi</th> </tr> </thead> <tbody> <tr> <td>30</td> <td>31</td> <td>0,00 ms</td> </tr> </tbody> </table> <p>Rute</p> <p>RIGHT - RIGHT - LEFT - UP - DOWN - RIGHT - LEFT - LEFT - DOWN - DOWN - RIGHT - LEFT - LEFT - UP - UP - RIGHT - RIGHT - DOWN - DOWN - RIGHT - LEFT - LEFT - LEFT - UP - UP - DOWN - RIGHT - RIGHT</p> <p>Kembali</p>	Langkah	Simpul diperiksa	Waktu Eksekusi	30	31	0,00 ms
Langkah	Simpul diperiksa	Waktu Eksekusi					
30	31	0,00 ms					
<p>Analisis: Penyelesaian ditemukan. Berdasarkan rute, pengguna berhasil kembali ke awal (KrustyKrab). Waktu eksekusi sedikit, tetapi langkah belum optimal.</p>							
Penyelesaian BFS							
 <pre>XRRXR TXXRX RRKRT RXTXX TRRRX</pre>	 <p>Hasil Pencarian</p> <p>Algoritma Breadth First Search (BFS) dengan TSP</p> <table border="1"> <thead> <tr> <th>Langkah</th> <th>Simpul diperiksa</th> <th>Waktu Eksekusi</th> </tr> </thead> <tbody> <tr> <td>20</td> <td>171</td> <td>2,00 ms</td> </tr> </tbody> </table> <p>Rute</p> <p>DOWN - DOWN - LEFT - LEFT - UP - UP - UP - DOWN - RIGHT - RIGHT - RIGHT - LEFT - LEFT - UP - DOWN - LEFT</p> <p>Kembali</p>	Langkah	Simpul diperiksa	Waktu Eksekusi	20	171	2,00 ms
Langkah	Simpul diperiksa	Waktu Eksekusi					
20	171	2,00 ms					
<p>Analisis: Penyelesaian ditemukan. Berdasarkan rute, pengguna berhasil kembali ke awal (KrustyKrab). Waktu eksekusi bertambah walau tidak signifikan, simpul diperiksa meningkat 6x lipat. Langkah merupakan langkah optimal.</p>							

Permasalahan 7	Penyelesaian DFS						
<pre>X T K R T X</pre>	<p>Hasil Pencarian</p> <p>Algoritma Depth First Search (DFS) dengan TSP</p> <table border="1"> <thead> <tr> <th>Langkah</th> <th>Simpul diperiksa</th> <th>Waktu Eksekusi</th> </tr> </thead> <tbody> <tr> <td>6</td> <td>7</td> <td>0,00 ms</td> </tr> </tbody> </table> <p>Rute RIGHT - RIGHT - LEFT - LEFT - LEFT - RIGHT</p> <p>Kembali</p>	Langkah	Simpul diperiksa	Waktu Eksekusi	6	7	0,00 ms
Langkah	Simpul diperiksa	Waktu Eksekusi					
6	7	0,00 ms					
Analisis: Penyelesaian ditemukan. Berdasarkan rute, pengguna berhasil kembali ke awal (KrustyKrab). Sekilas terlihat sama, namun langkah yang diambil tidak optimal. Hal ini dikarenakan alternatif prioritas pergerakan menuju ke kanan terlebih dahulu.							
Penyelesaian BFS							
	<p>Hasil Pencarian</p> <p>Algoritma Breadth First Search (BFS) tanpa TSP</p> <table border="1"> <thead> <tr> <th>Langkah</th> <th>Simpul diperiksa</th> <th>Waktu Eksekusi</th> </tr> </thead> <tbody> <tr> <td>4</td> <td>9</td> <td>1,00 ms</td> </tr> </tbody> </table> <p>Rute LEFT - RIGHT - RIGHT - RIGHT</p> <p>Kembali</p>	Langkah	Simpul diperiksa	Waktu Eksekusi	4	9	1,00 ms
Langkah	Simpul diperiksa	Waktu Eksekusi					
4	9	1,00 ms					
Analisis: Penyelesaian ditemukan. Berdasarkan rute, pengguna berhasil kembali ke awal (KrustyKrab). Langkah yang diambil optimal.							

3.4. Analisis Perbandingan Algoritma

Hal menarik yang diperoleh dari pengujian pada program adalah algoritma BFS hampir selalu menghasilkan langkah yang lebih sedikit dan efisien dibandingkan dengan algoritma DFS. [ada sisi lain, algoritma BFS selalu mengecek lebih banyak simpul dibandingkan algoritma DFS. Contohnya tertera pada Permasalahan 1 Tabel 3.3.1. BFS hanya membutuhkan 46 langkah, sementara DFS 113 langkah. Tetapi BFS mengecek 45918 simpul sementara DFS hanya 118 simpul.

Algoritma DFS merupakan algoritma yang menelusuri maze hingga ke ujung berdasarkan prioritas arah pergerakan, sedangkan algoritma BFS merupakan algoritma yang memprioritaskan pencarian ke setiap arah pergerakan. Berdasarkan pengujian pada tabel 3.3.1, terlihat bahwa algoritma DFS jauh lebih baik dalam kecepatan menentukan rute pencarian *treasure*.

Hal ini terjadi karena sebaran harta yang cenderung merata, sehingga setiap algoritma mau tidak mau harus menelusuri seluruh peta. Ini membuat algoritma BFS, yang mencari setiap rute, harus melakukan pencarian yang jauh lebih banyak dibandingkan dengan algoritma DFS. Berdasarkan hasil pengujian, diperoleh algoritma BFS lebih cocok untuk permasalahan yang tujuan pencarinya sedikit.

BAB V

KESIMPULAN DAN SARAN

5.1. Kesimpulan

Pada tugas besar II IF2211 Strategi Algoritma ini telah diimplementasikan algoritma BFS dan DFS beserta kelas-kelas pendukung dalam tujuan untuk menyelesaikan permasalahan *treasure maze* yang tertera pada spek. Kelas-kelas pendukung mencakup hal yang menangani GUI (Graphical User Interface) menggunakan *framework* Avalonia, kelas struktur data, dan kelas *interface*.

Pada tabel 3.3.1 dan 3.3.2. terlihat bahwa algoritma BFS dan DFS selalu berhasil menyelesaikan permasalahan yang valid. Algoritma BFS menampilkan hasil yang lebih optimal dibandingkan DFS. Walaupun demikian, BFS butuh alokasi memori yang banyak karena simpul yang dikunjungi lebih banyak secara signifikan. Persoalan TSP juga dapat dikomputasi secara BFS dan DFS dengan sifat yang sama dengan mencari *treasure*. Dengan demikian, penulis menyimpulkan bahwa melalui Tugas Besar II IF2211 Strategi Algoritma ini, dapat dibuat sebuah algoritma berbasis BFS dan DFS diimplementasikan dalam suatu program yang mengkomputasi penyelesaian *treasure maze*.

5.2. Saran

Tugas Besar II IF2211 Strategi Algoritma Semester II Tahun 2022/2023 menjadi salah satu tugas yang memberikan pelajaran baru bagi penulis. Berdasarkan pengalaman penulis mengerjakan tugas ini, berikut merupakan saran untuk pembaca yang ingin melakukan atau mengerjakan hal yang serupa.

1. Keefektifan dalam kerja sama tim merupakan hal yang penting dalam mengerjakan tugas ini. Selain pembagian tugas yang merata, penulis terbantu oleh beberapa kali kerja kelompok dan pemakaian *real-time collaboration app*, seperti Google Docs. Selain itu, penggunaan version control (Github) disarankan untuk memudahkan mengelola pekerjaan secara asinkron.
2. Perancangan algoritma dan struktur program perlu diperhatikan dalam mengerjakan tugas ini. Implementasi struktur program yang tepat akan memudahkan dalam pembuatan Graphical User Interface (GUI) dikarenakan memerlukan banyak *binding*.

5.3. Refleksi dan Tanggapan

Tugas Besar II IF2211 Strategi Algoritma merupakan salah satu tugas besar yang memerlukan usaha lebih dari biasanya. Meskipun demikian, kami mengapresiasi tugas besar ini karena memberikan kami kesempatan untuk bereksplorasi dan mengimplementasikan algoritma yang menarik.

DAFTAR PUSTAKA

Munir, R. (2022). *Strategi Algoritma: Breadth First Search (BFS) dan Depth First Search (DFS)* Bagian

1. Dilansir dari Homepage Rinaldi Munir:

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>. Diakses pada 24 Maret 2023.

Munir, R. (2022). Strategi Algoritma: Breadth First Search (BFS) dan Depth First Search (DFS)

Bagian 2. Dilansir dari Homepage Rinaldi Munir:

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

Diakses pada 24 Maret 2023.

LAMPIRAN

Repository Github

https://github.com/akbarmridho/Tubes2_mrKrrabs.git

Link Video Youtube

<https://youtu.be/LamfgSEn208>

CheckList Fitur

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa kesalahan	v	
2. Program berhasil running	v	
3. Program dapat membaca input	v	
4. Solusi yang diberikan program memenuhi	v	
5. Program dapat menampilkan visualisasi per rute pencarian	v	
6. Program berhasil menyelesaikan permasalahan TSP	v	