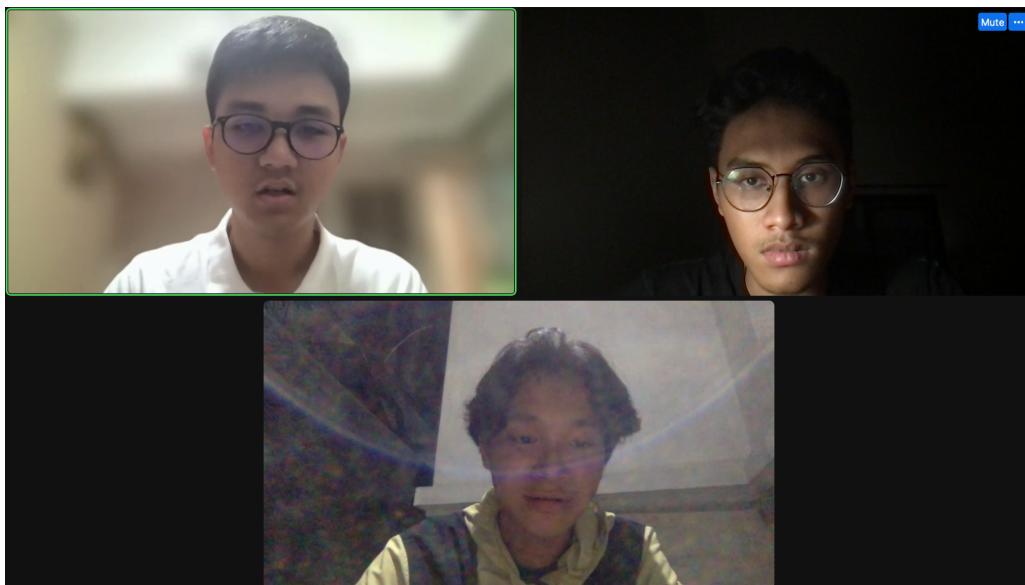


IF2211 Strategi Algoritma

IMPLEMENTASI ALGORITMA SEARCHING DFS DAN BFS DALAM PENYELESAIAN PERMASALAHAN MAZE TREASURE HUNT



Laporan Tugas Besar II

Disusun untuk memenuhi tugas mata kuliah IF2211 Strategi Algoritma pada
Semester 2 (dua) Tahun Akademik 2022/2023

Disusun oleh:

Enrique Alifio Ditya - 13521142

Ariel Jovananda - 13521086

Irsyad Nurwidianto Basuki - 135211072

K02

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

BANDUNG

2023

DAFTAR ISI

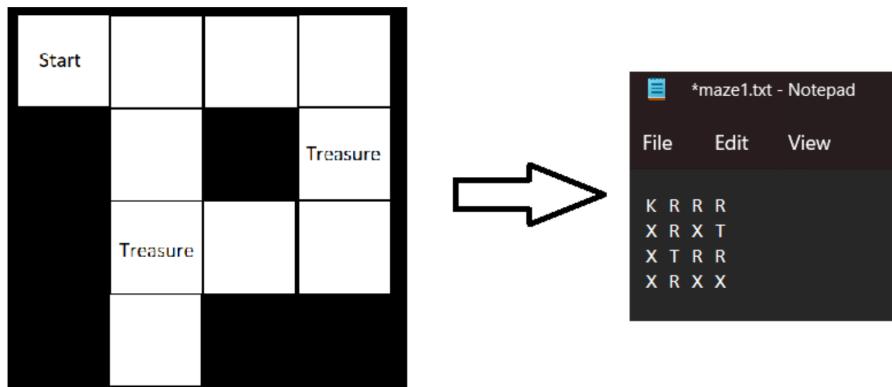
BAB I.....	1
BAB II.....	3
2.1 Dasar Teori.....	3
2.2 Penjelasan singkat mengenai C# desktop application development.....	7
BAB III.....	8
3.1 Langkah - Langkah pemecahan masalah.....	8
3.2 Pemetaan Persoalan dalam BFS dan DFS.....	9
BAB IV.....	11
4.1 Implementasi Program (pseudocode).....	11
4.2 Struktur Data dan Kelas Program.....	24
4.3 Tata Cara Penggunaan Program.....	28
4.4 Hasil Pengujian.....	30
4.5 Analisis.....	32
BAB V.....	35
5.1 Kesimpulan.....	35
5.2 Saran.....	35
5.3 Refleksi.....	35
5.4 Tanggapan.....	36
DAFTAR PUSTAKA.....	37
LAMPIRAN.....	38

BAB I

DESKRIPSI MASALAH

Dalam tugas besar ini, dibuatlah sebuah aplikasi dengan GUI sederhana yang dapat mengimplementasikan BFS dan DFS untuk mendapatkan rute dalam memperoleh seluruh *treasure* yang ada dalam suatu *maze*. Program dapat menerima dan membaca input sebuah file text 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 : Titik awal
- T : Treasure
- R : Grid yang mungkin diakses / sebuah lintasan
- X : Grid halangan yang tidak dapat diakses



Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS), kami 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. Kami juga diminta untuk memvisualisasikan input txt tersebut menjadi suatu grid maze serta hasil pencarian rute solusinya.

BAB II

LANDASAN TEORI

2.1 Dasar Teori

2.1.1 Graf Traversal

Graf Traversal adalah sebuah algoritma dalam bidang komputer yang digunakan untuk mengunjungi setiap simpul pada suatu graf dengan cara yang terorganisir. Tujuan dari algoritma traversal graf adalah untuk mencari solusi dari suatu masalah yang direpresentasikan dalam bentuk graf.

Algoritma traversal graf dapat digunakan untuk melakukan pencarian pada masalah yang tidak memiliki informasi tambahan (uninformed/blind search), seperti DFS, BFS, Depth Limited Search, Iterative Deepening Search, dan Uniform Cost Search. Selain itu, algoritma traversal graf juga dapat digunakan untuk melakukan pencarian pada masalah yang memiliki informasi atau heuristik (informed search), yaitu pencarian solusi yang mengetahui non-goal state yang lebih menjanjikan daripada yang lain, seperti Best First Search.

Proses pencarian solusi menggunakan traversal graf dapat dilakukan dengan dua pendekatan, yaitu graf statis dan graf dinamis. Pada graf statis, graf sudah terbentuk sebelum proses pencarian dilakukan sehingga graf direpresentasikan sebagai struktur data. Sedangkan pada graf dinamis, graf akan terbentuk saat proses pencarian dilakukan atau dalam kata lain graf tidak tersedia sebelum proses pencarian sehingga graf akan dibangun selama pencarian solusi.

Graf statis memiliki jumlah simpul dan sisi yang dapat ditetapkan. Biasanya parameter dari tipe data graf statik adalah kategori atau jenis graf. Kemudian, simpul dan sisi dapat ditambahkan secara opsional. Algoritma yang dapat digunakan untuk menelusuri jenis graf ini antara lain DFS, BFS, dan algoritma terkait lainnya.

DFS (Depth First Search) adalah algoritma traversal graf yang mengunjungi simpul-simpul pada graf secara mendalam, yaitu dengan mengeksplorasi salah satu simpul terdalam terlebih dahulu sebelum kembali ke simpul sebelumnya. Algoritma DFS biasanya menggunakan struktur data stack untuk mengimplementasikan pencarian.

BFS (Breadth First Search) adalah algoritma traversal graf yang mengunjungi simpul-simpul pada graf secara melebar, yaitu dengan mengeksplorasi semua simpul pada kedalaman yang sama sebelum melanjutkan ke simpul pada kedalaman berikutnya. Algoritma BFS biasanya menggunakan struktur data queue untuk mengimplementasikan pencarian.

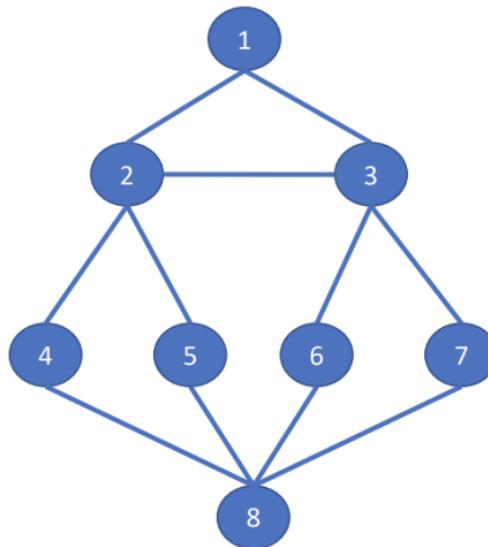
Dalam proses traversal graf, sangat penting untuk menghindari pengunjungan kembali pada simpul yang telah dikunjungi sebelumnya. Hal ini dapat diatasi dengan menggunakan struktur data yang tepat, seperti tabel hash atau array. Selain itu, perlu juga menetapkan kondisi berhenti pencarian agar algoritma traversal graf tidak terjebak dalam loop atau memeriksa simpul yang tidak perlu.

2.1.2 BFS

BFS (Breadth-First Search) adalah algoritma traversal graf yang digunakan untuk mengunjungi semua simpul pada sebuah graf secara melebar. Artinya, algoritma ini akan mengeksplorasi semua simpul yang berada pada jarak yang sama dari simpul awal terlebih dahulu sebelum melanjutkan ke simpul yang lebih jauh.

BFS biasanya menggunakan struktur data queue untuk mengimplementasikan pencarian. Pada awalnya, simpul awal dimasukkan ke dalam queue. Kemudian, simpul yang bersebelahan langsung dengan simpul awal (disebut dengan "tetangga") dimasukkan ke dalam queue. Setelah itu, simpul awal dihapus dari queue dan proses pengunjungan simpul dilanjutkan dengan

mengambil simpul pertama dari queue dan mengecek tetangganya. Proses ini berulang terus menerus hingga semua simpul pada graf terkunjungi. Berikut adalah contoh implementasi penggunaan BFS:



Gambar 2.1 sumber: DFS dan BFS, Rinaldi Munir

Langkah Algoritma yang dilakukan adalah sebagai berikut:

1. Mengunjungi simpul paling atas atau paling utama
2. Setelah itu pencarian dilakukan dengan cara melakukan ke tetangganya terlebih dahulu sebelum melakukan pencarian ke simpul anak.

Sehingga secara sederhana urutan penelusuran dilakukan secara berurut dengan urutan:

1 – 2 – 3 – 4 – 5 – 6 – 7 – 8

2.1.3 DFS

DFS (Depth First Search) adalah algoritma traversal graf yang digunakan untuk mengunjungi setiap simpul pada suatu graf secara mendalam atau vertikal, yaitu dengan mengeksplorasi salah satu simpul terdalam terlebih dahulu sebelum kembali ke simpul sebelumnya. Dalam DFS, pencarian dilakukan dengan mengunjungi simpul yang bertetangga langsung dengan simpul saat ini, kemudian melanjutkan ke simpul tetangga dari simpul tetangga tersebut, dan seterusnya hingga seluruh simpul dalam graf tercakup.

Proses pencarian DFS dapat diimplementasikan dengan menggunakan struktur data stack atau rekursi. Dalam implementasi menggunakan stack, simpul pertama kali dimasukkan ke dalam stack, kemudian simpul yang bertetangga dengan simpul tersebut dimasukkan ke dalam stack dan diunjungi satu per satu. Proses ini berlanjut hingga seluruh simpul dalam graf tercakup. Pada implementasi menggunakan rekursi, simpul saat ini dikunjungi dan fungsi rekursif dipanggil untuk setiap simpul bertetangga dengan simpul tersebut.

Berikut adalah langkah-langkah cara melakukan DFS:

1. Pilih simpul awal dan tandai sebagai dikunjungi.
2. Kunjungi simpul tetangga yang belum dikunjungi secara berurutan.
3. Setiap kali mengunjungi simpul tetangga, tandai sebagai dikunjungi.
4. Ulangi langkah 2-3 sampai semua simpul telah dikunjungi atau sampai kriteria berhenti tercapai.
5. Jika masih ada simpul yang belum dikunjungi, kembali ke simpul sebelumnya dan ulangi langkah 2-4 untuk simpul lainnya.
6. Ulangi langkah 5 sampai semua simpul telah dikunjungi.

2.2 Penjelasan singkat mengenai C# desktop application development

C# adalah bahasa pemrograman berorientasi objek yang populer yang dikembangkan oleh Microsoft. C# banyak digunakan untuk mengembangkan aplikasi desktop, aplikasi web, dan game. Dalam tugas besar ini, kita akan fokus pada pengembangan aplikasi desktop menggunakan C#.

C# menawarkan berbagai fitur yang dapat digunakan untuk mengembangkan aplikasi desktop, termasuk Windows Forms, WPF, dan UWP. Windows Forms adalah cara tradisional untuk mengembangkan aplikasi desktop dalam C#, sedangkan WPF dan UWP menawarkan pendekatan yang lebih modern dan memungkinkan pembuatan antarmuka pengguna yang lebih canggih. Kini, pembuatan aplikasi C# dapat dibantu dengan berbagai alat bantu, seperti Visual Studio, .NET Framework dan lain sebagainya. Dengan alat bantu ini, pengembang dapat mengembangkan aplikasi dengan kode yang sama untuk berbagai platform.

BAB III

PEMECAHAN MASALAH

3.1 Langkah - Langkah pemecahan masalah

Permasalahan yang akan diselesaikan adalah *treasure hunt*. Program akan membaca sebuah *input file* dan mengubahnya menjadi matriks berisi angka 0, 1, dan 2. Angka 0 menunjukkan jalur yang dapat dilalui, angka 1 menunjukkan jalur yang tidak dapat dilalui, serta angka 2 menunjukkan bahwa terdapat *treasure* pada suatu jalur atau *node*.

Pencarian dilakukan dengan cara mulai dari titik awal, yaitu K, dan mencari seluruh treasure, yaitu T, yang ada dalam *maze*. Tiap jalur yang akan ditelusuri dapat direpresentasikan sebagai *tree*. Jalur - jalur dari *tree* disebut sebagai *node*. Titik awal pencarian (*root tree*) dapat memiliki satu anak ataupun beberapa anak untuk ditelusuri. Penelusuran dilakukan dengan metode iteratif dan untuk setiap node yang akan ditelusuri akan dicatat pada struktur data *queue* untuk algoritma BFS sedangkan dicatat pada struktur data *stack* untuk algoritma DFS.

Secara umum, algoritma BFS untuk mencari seluruh treasure di dalam maze dilakukan dengan cara sebagai berikut:

1. Inisialisasi queue, visited, dan list of treasure nodes.
2. Buat node awal dan masukkan ke dalam queue.
3. Selama queue tidak kosong, lakukan hal berikut:
 - a. Ambil node pertama dari queue.
 - b. Periksa apakah node tersebut merupakan treasure. Jika ya, tambahkan ke dalam list of treasure nodes dan periksa apakah seluruh treasure sudah ditemukan. Jika seluruh treasure sudah ditemukan, keluarkan path yang dilalui dari node awal ke

- treasure terakhir, dan berhentilah pencarian. Jika belum, kosongkan queue dan visited, dan mulai pencarian dari awal lagi.
- c. Tandai node tersebut sebagai visited.
 - d. Buat semua node tetangga dari node tersebut yang belum visited dan tambahkan ke dalam queue.
4. Jika pencarian selesai dan seluruh treasure belum ditemukan, keluarkan pesan bahwa semua treasure tidak ditemukan.
 5. Jika pencarian selesai dan seluruh treasure sudah ditemukan, keluarkan path yang dilalui dari node awal ke treasure terakhir.

Sementara itu, algoritma DFS mengikuti langkah umum yang sama, namun hanya mengganti struktur data *queue* dengan *stack* dalam pencatatan kandidat node yang akan ditelusuri.

Permasalahan lain yang akan diselesaikan adalah *Graphical User Interface* (GUI). GUI digunakan untuk memvisualisasikan langkah - langkah serta solusi dari penggunaan algoritma pencarian dalam *maze*. GUI juga dapat digunakan untuk menerima file .txt tertentu yang ingin digunakan secara lebih mudah. Permasalahan ini diselesaikan dengan melakukan pengembangan aplikasi *desktop* yang menggunakan .NET Framework dalam penggerjaannya.

3.2 Pemetaan Persoalan dalam BFS dan DFS

3.2.1 BFS

Pada pencarian dengan algoritma BFS, digunakan struktur data *queue* untuk melacak *node* yang perlu dikunjungi selanjutnya. Pencarian dimulai dengan menambahkan node awal ke dalam queue. Algoritma kemudian berulang kali melakukan dequeues pada node, mengunjungi node tersebut, dan enqueue tetangga-tetangga yang belum dikunjungi ke queue.

Pencarian BFS ini melacak node yang sudah dikunjungi dengan menggunakan array 2D yang disebut "visited". Jika sebuah node telah dikunjungi (ditandai dengan angka 1), tidak akan menambahkannya kembali ke queue. Pencarian BFS berakhir ketika menemukan semua harta karun di dalam labirin atau ketika telah mengunjungi semua node di labirin.

3.2.2 DFS

Pada pencarian dengan algoritma DFS, digunakan struktur data *stack* untuk melacak jalur saat ini yang sedang dijelajahi. Pertama, *node* awal ditambahkan ke dalam *stack*. Kemudian, memasuki loop yang terus berlanjut selama masih terdapat node pada *stack*. Setiap kali dilakukan iterasi, melakukan *pop* node memeriksa apakah itu adalah sebuah harta karun. Jika node tersebut adalah harta karun dan belum pernah dikunjungi sebelumnya, maka harta karun tersebut ditambahkan ke dalam daftar harta karun yang ditemukan. Jika semua harta karun sudah ditemukan, pencarian berhenti dan mengembalikan jalur yang diambil. Jika tidak, *stack* dikosongkan dan memulai pencarian baru dari node awal.

Node yang telah dikunjungi akan dilacak untuk menghindari menjelajahi node yang sama beberapa kali. Hal ini dilakukan menggunakan array 2D yang disebut "visited". Jika suatu node telah dikunjungi, maka elemen yang sesuai dalam array tersebut diatur menjadi 1.

BAB IV

ANALISIS PEMECAHAN MASALAH

4.1 Implementasi Program (*pseudocode*)

1. MainPage.xaml.cs

```
// Define MainPage class
class MainPage:
    function MainPage():
        // Initialize component
        InitializeComponent()
        // Define class variables
        String filePath = null
        Int [,] maze = null
        Int sliderValue = null
        Grid grid = null
        FileResult result = null

        // Define function to load maze file
        function baseMaze(result):
            try:
                // Load the maze from the file
                maze ← Maze(0, 0)
                maze.Load(result.FullPath)

                // Extract data from maze and save it to matrix
                fileName.Text ← $"\"{result.FileName}\\""
                filePath ← result.FullPath

                matrix ← create 2D array with size (maze.Rows,
maze.Cols)
                startcount ← 0
                for i in range(matrix.GetLength(0)):
                    for j in range(matrix.GetLength(1)):
                        if maze[i, j] = 1:
                            matrix[i, j] ← -1
                        else if maze[i, j] = 0 and i = maze.StartRow and
j = maze.StartCol and startcount = 0:
```

```

        matrix[i, j] ← 0
        Startcount = Startcount + 1
    else if maze[i, j] = 2:
        matrix[i, j] ← 2
    else if maze[i, j] = 0:
        matrix[i, j] ← 1

    // Print matrix to console
    for i in range(matrix.GetLength(0)):
        for j in range(matrix.GetLength(1)):
            print(matrix[i, j] + " ")
        print()

    // Update class variable and display the maze
    this.maze ← matrix
    await DisplayMaze(matrix, maze.Rows, maze.Cols)
except InvalidMazeFormatException as ex:
    // Handle exception if maze file has invalid format
    await DisplayAlert("Error", ex.Message.ToString(), "OK")
    return

// Define function to handle file picker button click
function Button_Clicked(sender, e):
    // Open file picker and load the selected file if it is a
    .txt file
    options ← PickOptions()
    try:
        result ← await FilePicker.Default.PickAsync(options)
        if result != null:
            if result.FileName.EndsWith("txt",
StringComparison.OrdinalIgnoreCase):
                baseMaze(result)
            else:
                throw new ArgumentException("Invalid file
extension. Only .txt files are allowed")
        except ArgumentException as ex:
            // Handle exception if file picker is canceled or an
            invalid file is selected
            await DisplayAlert("Error", ex.Message, "OK")

```

```

    return

// Define function to display the maze
function DisplayMaze(matrix, numberOfRows, numberOfColumns):
    // Create a new grid to hold the maze
    rowDefinitions = new RowDefinitionCollection()
    grid ← Grid(ColumnSpacing ← 1, RowSpacing ← 1,
    BackgroundColor ← Color.FromRgb(255, 255, 255))

    // Define row and column definitions
    for i in range(numberOfRows):
        rowDefinitions.Add(RowDefinition { Height = new
    GridLength(1, GridUnitType.Star) })
        columnDefinitions = new ColumnDefinitionCollection()
        for i in range(numberOfColumns):
            columnDefinitions.Add(ColumnDefinition { Width = new
    GridLength(1, GridUnitType.Star) })

    // Set the RowDefinitions and ColumnDefinitions for the grid
    grid.RowDefinitions ← rowDefinitions
    grid.ColumnDefinitions ← columnDefinitions

    // Loop through the matrix and create a Label for each
element
    for i ← 0 to matrix row count
    for j ← 0 to matrix column count
        label ← new Label
        if matrix[i, j] = 2 or matrix[i, j] = 5
            label.Text = "Treasure"
        else if matrix[i, j] = 0 or matrix[i, j] = 3
            label.Text ← "Start"
        else
            label.Text ← ""
        if matrix[i, j] = 0 or matrix[i, j] = 2 or matrix[i, j] = 3
        or matrix[i, j] = 5
            label.TextColor ← Color.FromRgb(0, 0, 0)
        else
            label.TextColor ← Color.FromRgb(255, 255, 255)
        if matrix[i, j] = -1

```

```

        label.BackgroundColor ← Color.FromRgb(0, 0, 0)
    else if matrix[i, j] = 3 or matrix[i, j] = 4 or matrix[i, j]
= 5
        label.BackgroundColor ← Color.FromRgb(0, 255, 0)
else
        label.BackgroundColor ← Color.FromRgb(255, 255, 255)
label.HorizontalTextAlignment ← TextAlignment.Center
label.VerticalTextAlignment ← TextAlignment.Center
Grid.SetRow(label, i)
Grid.SetColumn(label, j)
grid.Children.Add(label)
mazeView.Content ← grid
this.grid ← grid
Task.Delay(sliderValue)

Function visu_Clicked() {
stopwatch ← new Stopwatch
steps ← 0
maze ← new Maze(0, 0)
try
    maze.Load(filePath)
catch FileNotFoundException
    DisplayAlert("Error", "Please select a file.", "OK")
    return
catch Exception ex
    DisplayAlert("Error", ex.Message.ToString(), "OK")
    return
if bfsCheckBox.IsChecked and dfsCheckBox.IsChecked
    DisplayAlert("Error", "Please select only 1 checkbox to run the
algorithm.", "OK")
    stopwatch.Start()
    stopwatch.Stop()
else if bfsCheckBox.IsChecked
    bfs ← new BFS(maze)
    stopwatch.Start()
    if tspCheckBox.IsChecked
        bfsPath ← bfs.SearchTreasures(maze.StartRow, maze.StartCol,
true)
        stopwatch.Stop()

```

```

else
    bfsPath ← bfs.SearchTreasures(maze.StartRow, maze.StartCol,
false)
    stopwatch.Stop()
visualization(bfsPath, bfsPath)
steps ← bfsPath.Count
if bfsPath.Count() > 0
    routeInfo.Text ← $"Route :
{bfsPath.Last().GetDirections("")}"
    stepsInfo.Text ← $"Steps : {bfsPath.Count() - 1}"
else
    routeInfo.Text ← "Route : -"
    stepsInfo.Text ← "Steps : 0"
else if (dfsCheckBox.IsChecked)
    // Run DFS algorithm
    stopwatch.Start()
    dfs ← new DFS(maze)
    cleanDFS ← new DFS(maze)
    if (tspCheckBox.IsChecked)
        dfsPath ← dfs.SearchTreasures(maze.StartRow,
maze.StartCol, true)
        cleanDFSPath ← cleanDFS.SearchTreasures(maze.StartRow,
maze.StartCol, true, true)
        stopwatch.Stop()
    else
        dfsPath ← dfs.SearchTreasures(maze.StartRow,
maze.StartCol, false)
        cleanDFSPath ← cleanDFS.SearchTreasures(maze.StartRow,
maze.StartCol, false, true)
        stopwatch.Stop()

    // visualize the path and update steps counter
    visualization(dfsPath, cleanDFSPath)
    steps ← dfsPath.Count()

    // display the route and steps information
    if (cleanDFSPath.Count() > 0) {
        routeInfo.Text ← $"Route :

```

```

{cleanDFSPath.Last().GetDirections("")}"
    stepsInfo.Text ← $"Steps : {cleanDFSPath.Count() - 1}"
} else {
    routeInfo.Text ← "Route : -"
    stepsInfo.Text ← "Steps : 0"
}
} else {
    // No checkbox selected, show error message
    stopwatch.Start()
    stopwatch.Stop()
    await DisplayAlert("Error", "Please select a checkbox to run
the algorithm.", "OK")
    return
}

// display execution time and steps information
executionTime.Text ← $"Execution Time :
{stopwatch.ElapsedMilliseconds} ms"
nodesCounter.Text ← $"Nodes : {steps}"
SemanticScreenReader.Announce(nodesCounter.Text)
SemanticScreenReader.Announce(executionTime.Text)
SemanticScreenReader.Announce(routeInfo.Text)
SemanticScreenReader.Announce(stepsInfo.Text)

Function mazeSlider(sender, e) :
    value ← e.NewValue
    sliderValue ← 1000 - value

Function ResetButton(sender, e) :
    // reset the maze and clear all information
    mazeView.Content ← null
    executionTime.Text ← $"Execution Time : "
    nodesCounter.Text ← $"Nodes : "
    routeInfo.Text ← "Route : "
    stepsInfo.Text ← "Steps : "
    dfsCheckBox.IsChecked ← false
    bfsCheckBox.IsChecked ← false

```

```

tspCheckBox.IsChecked ← false
if (this.filePath != null)
    baseMaze(result)

function visualization(list, cleanedList):
    // Initialize a matrix with all elements set to -1
for i in range(matrix.GetLength(0)):
        for j in range(matrix.GetLength(1)):
            matrix[i, j] = -1;
prevVal ← 0
count ← 0
index ← 0

// Traverse through the List of nodes
while index <= list.length:
    if count > 0:
        if prevVal > 0:
            // Increment the previous value in the matrix
            prevVal = prevVal + 1
            matrix[list[index - 1].X, list[index - 1].Y] ←
prevVal
        else:
            // Increment the value in the matrix
            matrix[list[index - 1].X, list[index - 1].Y]++

        // Change the color of the tile at the previous node
        ChangeTileColor(matrix[list[index - 1].X, list[index - 1].Y], list[index - 1].X, list[index - 1].Y)

        // Increment the index
        if index = list.length:
            index++

    // Searching for the current index in the list
    if index < list.length:
        if matrix[list[index].X, list[index].Y] = -1:
            prevVal ← 0
            matrix[list[index].X, list[index].Y]++

```

```

else:
    prevVal ← matrix[list[index].X, list[index].Y]
    matrix[list[index].X, list[index].Y] ← 0

    // Change the color of the tile at the current node
    ChangeTileColor(0, list[index].X, list[index].Y)

    // Increment the index
    index++

    if count == 0:
        count++

index = 0

// Traverse through the cleaned list of nodes
while index < cleanedList.length:
    if maze[cleanedList[index].X, cleanedList[index].Y] = 0:
        maze[cleanedList[index].X, cleanedList[index].Y] ← 3
    else if maze[cleanedList[index].X, cleanedList[index].Y] = 1:
        maze[cleanedList[index].X, cleanedList[index].Y] ← 4
    else if maze[cleanedList[index].X, cleanedList[index].Y] = 2:
        maze[cleanedList[index].X, cleanedList[index].Y] ← 5

    // Increment the index
    index++

// Display the maze
DisplayMaze(maze, maze.GetLength(0), maze.GetLength(1))

```

2. BFS.cs

```

Function searchTreasures(maze):
    Queue<Node> queue ← new Queue<Node>();
    Node startNode ← new Node(startX, startY, null);
    List<Node> fullPath ← new List<Node>();
    queue.Enqueue(startNode);

```

```

while queue is not empty:
    currentNode ← dequeue from queue

    if currentNode is a treasure and not previously visited:
        mark currentNode as visited
        add currentNode to treasureNodes list
        if all treasures have been found:
            if tsp is true:
                fullPath ← concatenate fullPath with
SearchPathBack(currentNode)
            else:
                fullPath ← concatenate fullPath with
currentNode.ListPath()

            return fullPath
        else:
            reset visited array and clear queue

        mark currentNode as visited
        nextMoves ← generate all possible next moves from
currentNode
        for each nextMove in nextMoves:
            if nextMove has not been visited:
                enqueue nextMove into queue

    if no treasures found:
        print "All treasures not found."
    else:
        if tsp is true:
            fullPath ← concatenate fullPath with SearchPathBack(last
treasureNode)
        else:
            fullPath ← concatenate fullPath with last
treasureNode.ListPath()

        print "Treasures Found: " number of treasureNodes
print "Path taken: " fullPath[-1].GetDirections("")
```

```

return fullPath

function SearchPathBack(startNode):
    Queue<Node> queue = new Queue<Node>();
    visited = new int[maze.Rows, maze.Cols];
    Node currentNode = startNode;
    queue.Enqueue(startNode);

    while queue is not empty:
        currentNode ← dequeue from queue

        if currentNode is the start node:
            return currentNode.ListPath()

        mark currentNode as visited
        nextMoves ← generate all possible next moves from
        currentNode
        for each nextMove in nextMoves:
            if nextMove has not been visited:
                enqueue nextMove into queue

    return currentNode.ListPath()

```

3. DFS.cs

```

function search_treasures(self, start_x, start_y, tsp=False,
reset=False):
    stack = Stack()
    start_node = Node(start_x, start_y, None)
    full_path = []
    stack.push(start_node)

    print("DFS begins search...")

    while len(stack) > 0:
        current_node = stack.pop()

```

```

    if not reset:
        full_path.append(current_node)

        if self.maze[current_node.x, current_node.y] == 2 and
current_node not in self.treasure_nodes:
            print(f"Treasure found at ({current_node.x},
{current_node.y})!")
            self.treasure_nodes.append(current_node)

        if len(self.treasure_nodes) == self.maze.treasures:
            print()
            print(f"Treasures Found: {len(self.treasure_nodes)}")

        if tsp:
            if reset:
                full_path +=
self.search_path_back(current_node)
            else:
                full_path +=
self.tsp_with_backtrack(current_node)
            elif reset:
                full_path += current_node.list_path()

            print("Path taken: ")
            print(full_path[-1].get_directions(""))
            return full_path
        elif reset:
            self.visited = [[0 for _ in range(self.maze.cols)]]
for _ in range(self.maze.rows)]
            stack.clear()

            self.visited[current_node.x][current_node.y] = 1

            next_moves = current_node.generate_next_moves(self.maze)

            neighbours = 0

            for next_move in next_moves:

```

```

        if not self.visited[next_move.x][next_move.y]:
            stack.push(next_move)
            neighbours += 1

    if not reset and neighbours == 0 and current_node.parent:
        stack.push(current_node.parent)

    if len(self.treasure_nodes) == 0:
        print("All treasures not found.")
    else:
        print()
        print(f"Treasures Found: {len(self.treasure_nodes)}")

    if tsp:
        if reset:
            full_path +=
self.search_path_back(self.treasure_nodes[-1])
        else:
            full_path +=
self.tsp_with_backtrack(self.treasure_nodes[-1])
        elif reset:
            full_path += self.treasure_nodes[-1].list_path()

        print("Path taken: ")
        print(full_path[-1].get_directions(""))

    return full_path

function search_path_back(self, start_node):
    stack = Stack()
    self.visited = [[0 for _ in range(self.maze.cols)] for _ in
range(self.maze.rows)]
    current_node = start_node
    stack.push(start_node)

    print()
    print("DFS traces way back...")
    print()

```

```

while len(stack) > 0:
    current_node = stack.pop()

    if current_node.x == self.maze.start_row and current_node.y
    == self.maze.start_col:
        return current_node.list_path()
    else:
        self.visited[current_node.x][current_node.y] = 1
        next_moves = current_node.generate_next_moves(self.maze)

        for next_move in next_moves:
            if not self.visited[next_move.x][next_move.y]:
                stack.push(next_move)

return current_node.list_path()

function TSPWithBacktrack(startNode):
    stack = new Stack<Node>()
    fullPath = new List<Node>()
    currentNode = startNode
    stack.push(startNode)
    first = true

    while stack is not empty:
        currentNode = stack.pop()

        if first:
            first = false
        else:
            fullPath.add(currentNode)

        if currentNode.X is equal to maze.StartRow and currentNode.Y
        is equal to maze.StartCol:
            return fullPath
        else:
            visited[currentNode.X, currentNode.Y] = 1
            nextMoves = currentNode.GenerateNextMoves(maze)
            neighbours = 0

```

```

    for nextMove in nextMoves:
        if visited[nextMove.X, nextMove.Y] is not equal to 1:
            stack.push(nextMove)
            neighbours = neighbours + 1

        if neighbours is equal to 0 and currentNode.Parent is not
equal to null:
            stack.push(currentNode.Parent)

    return fullPath

```

4.2 Struktur Data dan Kelas Program

Program ini diimplementasikan dengan pendekatan berorientasi objek yang mengandung beberapa kelas dan struktur data sebagai pendukung pencarian *treasure*. Berikut adalah kelas-kelas dan struktur data yang digunakan.

4.2.1 Queue

Struktur data *queue* ini digunakan dalam algoritma pencarian BFS untuk menyimpan node yang belum diproses secara FIFO (First-In, First-Out).

4.2.2 Stack

Struktur data *stack* ini digunakan dalam algoritma pencarian DFS untuk menyimpan *node* yang belum diproses secara LIFO (Last-In, First-Out).

4.2.3 List

Struktur data *list* ini digunakan untuk menyimpan *node* dalam sebuah *list* atau bisa dikenal dengan array yang berisi *list*.

4.2.4 Maze

Kelas Maze berisi logika untuk merepresentasikan maze dalam bentuk grid 2D dan berbagai operasi terkait maze, seperti pembacaan maze dari file, tampilan maze pada konsol, dan pengambilan properti seperti jumlah baris, jumlah kolom, dan jumlah harta karun di dalam maze.

Kelas ini memiliki beberapa properti dan metode, antara lain:

1. `_grid`: array 2D yang menyimpan sel-sel di dalam maze.
2. `_rows` dan `_cols`: jumlah baris dan kolom dalam maze.
3. `_startRow` dan `_startCol`: posisi awal (start point) dalam maze.
4. `_treasures`: jumlah harta karun dalam maze.
5. `Maze(rows, cols)`: konstruktor untuk membuat maze kosong dengan jumlah baris dan kolom tertentu.
6. `Load(filePath)`: metode untuk membaca maze dari file dengan format tertentu dan memuatnya ke dalam `_grid`.
7. `Display()`: metode untuk menampilkan maze pada konsol.
8. `Rows, Cols, StartRow, StartCol, dan Treasures`: properti untuk mengambil dan mengubah nilai properti terkait maze.

Selain itu, kelas ini juga memiliki kelas turunan `InvalidMazeFormatException` yang merupakan pengecualian yang dilemparkan ketika ada kesalahan dalam format file maze yang dibaca oleh metode `Load`.

4.2.5 Node

Kelas Node merupakan representasi dari sebuah titik atau posisi dalam sebuah grid atau peta yang digunakan untuk mencari jalur atau pathfinding. Kelas ini memiliki tiga atribut yaitu `x` dan `y` yang merepresentasikan koordinat titik pada grid, dan `parent` yang merupakan referensi ke objek Node lain yang merupakan titik sebelumnya pada jalur yang telah dicari. Kelas Node

memiliki beberapa method seperti GenerateNextMoves yang digunakan untuk menghasilkan daftar node yang bisa dicapai dari node saat ini, GetDirections yang digunakan untuk menghasilkan string yang merepresentasikan jalur dari titik awal hingga titik saat ini, dan ListPath yang menghasilkan daftar node yang merepresentasikan seluruh jalur yang sudah ditemukan dari titik awal hingga titik saat ini.

4.2.6 Solver

Kelas Solver adalah sebuah kelas abstract yang merupakan blueprint untuk kelas-kelas solver yang akan diimplementasikan untuk menyelesaikan maze. Kelas ini memiliki tiga atribut yaitu maze, visited, dan treasureNodes. Atribut maze adalah objek Maze yang akan disolve, visited adalah array 2 dimensi yang merekam sel-sel yang telah dikunjungi oleh algoritma, dan treasureNodes adalah list dari objek Node yang merepresentasikan posisi dari semua treasure yang ada di maze.

Kelas Solver memiliki dua method abstract yaitu SearchTreasures dan SearchPathBack. Method SearchTreasures berfungsi untuk mencari semua treasure yang ada di dalam maze, sedangkan method SearchPathBack berfungsi untuk menemukan jalur dari posisi saat ini kembali ke posisi awal di dalam maze.

Kelas Solver sendiri merupakan superclass dari kelas solver yang akan diimplementasikan, seperti BFS atau DFS, dan digunakan untuk mengurangi duplikasi kode antar solver.

4.2.7 BFS

Kelas BFS merupakan turunan dari kelas Solver yang mengimplementasikan algoritma Breadth-First Search (BFS) untuk mencari semua harta karun yang dapat dicapai dari posisi awal yang ditentukan.

Konstruktor kelas BFS menerima objek Maze yang akan digunakan dalam pencarian. Metode SearchTreasures() dan SearchPathBack() diimplementasikan untuk mengambil objek Node startNode dan mengembalikan List<Node> yang berisi jalur yang ditempuh dari startNode ke harta karun atau ke titik awal, sesuai dengan fungsi masing-masing.

Dalam metode SearchTreasures(), pencarian dimulai dari posisi startNode dengan membuat sebuah Queue<Node> untuk menyimpan node yang akan diperiksa. Setiap kali node diambil dari queue, BFS memeriksa apakah node tersebut adalah harta karun, dan jika iya, node tersebut ditambahkan ke dalam List<Node> treasureNodes. Setelah semua harta karun ditemukan, BFS mengembalikan jalur terpendek yang melewati semua harta karun tersebut jika parameter tsp bernilai true, atau hanya jalur dari startNode ke harta karun terakhir jika tsp bernilai false.

Dalam metode SearchPathBack(), BFS menelusuri jalur kembali dari harta karun terakhir ke titik awal menggunakan BFS. Metode ini mengembalikan jalur terpendek dari harta karun terakhir ke titik awal.

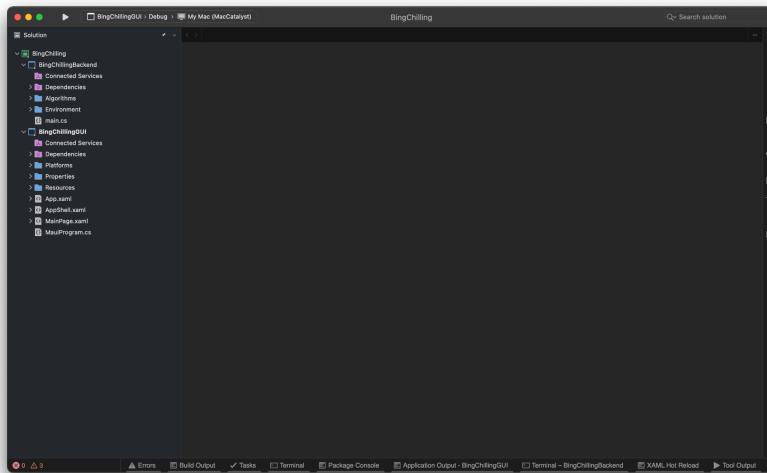
4.2.8 DFS

Kelas DFS merupakan turunan dari kelas Solver yang mengimplementasikan algoritma Depth-First Search (DFS) untuk mencari semua harta karun yang dapat dicapai dari posisi awal yang ditentukan.

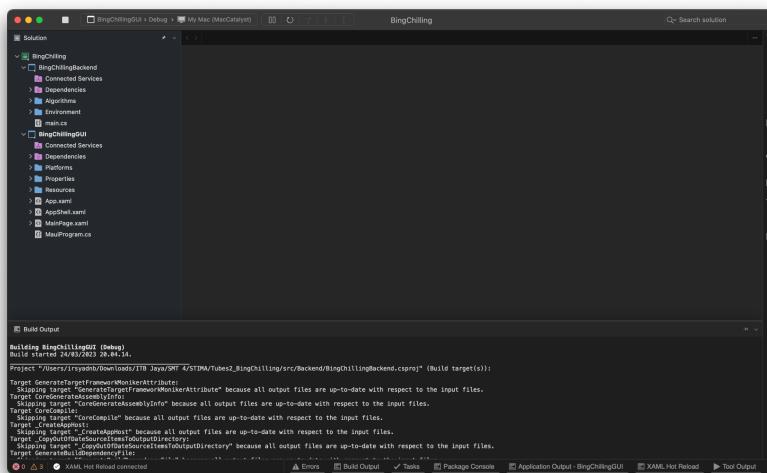
Sama seperti BFS, kelas DFS memiliki metode SearchTreasures dan SearchPathBack yang diimplementasikan untuk pencarian harta karun dalam maze. Bedanya, proses pencarian menggunakan pencatatan node yang didukung oleh struktur data stack, sebagaimana dijelaskan sebelumnya. Metode-metode yang dimiliki oleh kelas ini merupakan bentuk polimorfisme dari kelas BFS yang dijelaskan sebelumnya.

4.3 Tata Cara Penggunaan Program

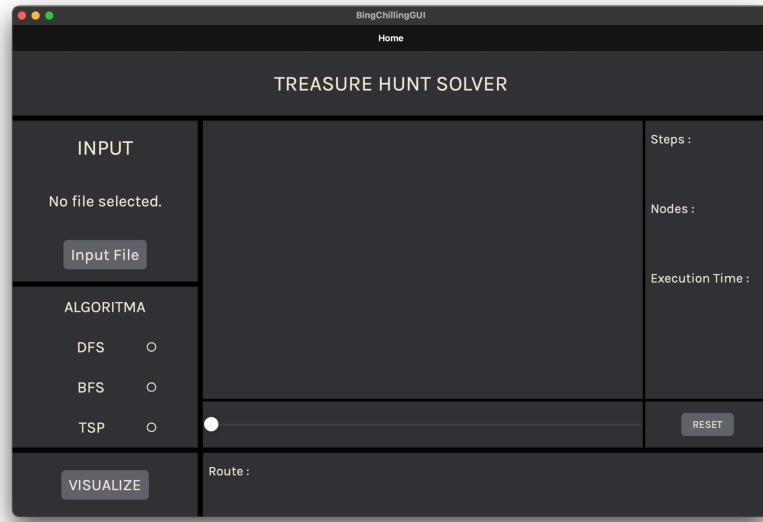
4.3.1 Cara menjalankan program



Gambar 4.3.1.1



Gambar 4.3.1.2



Gambar 4.3.1.3

Untuk menjalankan program, buka file Visual Studio bernama “BingChilling.sln” pada folder *src* seperti Gambar 4.1. Selanjutnya build projek BingChillingGui dan pilih platform yang diinginkan. Project akan ter-build seperti Gambar 4.2. Terakhir GUI aplikasi BingChillingGUI akan muncul seperti Gambar 4.3 dan program siap untuk digunakan.

4.3.2 Cara menggunakan program

Untuk menjalankan program ini, user perlu memasukkan beberapa *input* sebagai berikut:

1. File .txt

User dapat memilih file txt yang mengandung peta untuk dimasukkan ke program.

2. Pemilihan BFS atau DFS dan opsi TSP

User dapat melakukan cek pada *checkbox* untuk memilih algoritma pencarian. User juga dapat memilih apakah ingin memilih opsi TSP sekaligus saat memilih memilih algoritma pencarian.

3. Tombol *visualize*

User dapat menekan tombol *visualize* untuk memulai algoritma pencarian serta menampilkan langkah - langkah dan solusi dari hasil pencarian. Tombol ini hanya dapat digunakan apabila *checkbox* dari BFS atau DFS telah ditekan. Apabila tidak ada atau kedua tombol pilihan algoritma yang diterima, akan mengeluarkan *error*.

4. Slider

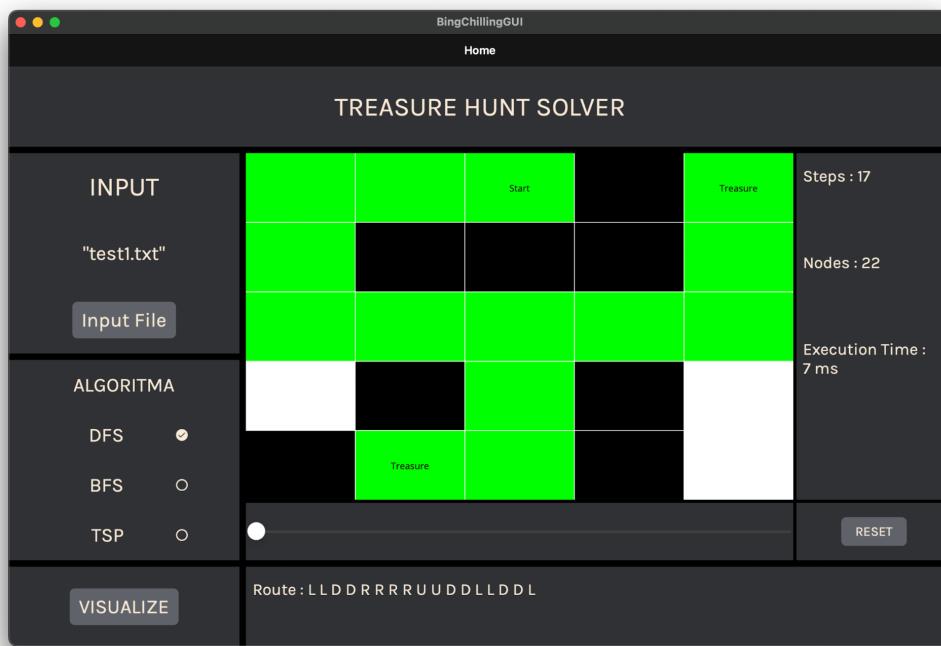
User dapat mengatur kecepatan dari tiap langkah pencarian dengan menggeser *slider* yang telah disediakan dibawah maze dengan kondisi semakin kanan semakin cepat.

5. Tombol *reset*

User dapat menekan tombol *reset* tiap ingin mengganti percobaan algoritma pencarian, User juga dapat menekan tombol *reset* saat ingin mengganti file txt yang digunakan.

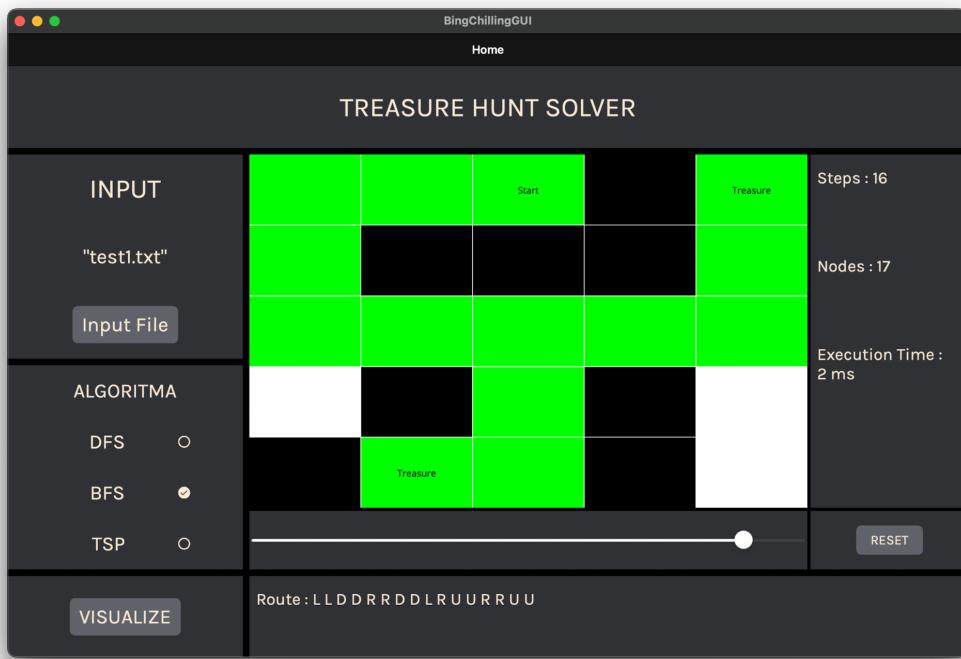
4.4 Hasil Pengujian

Pengujian algoritma BFS, DFS serta pengaplikasian TSP dilakukan dengan file “test1.txt”. Pemasukan file akan langsung menampilkan maze sesuai isi file tersebut. Program akan menampilkan lokasi harta karun bertuliskan “treasure”, lokasi mulai bertuliskan “start”, jalan yang dapat dilalui dengan kotak putih serta tembok atau jalan yang tidak dapat dilalui dengan kotak hitam. Saat pencarian dilakukan, tiap langkah pencarian akan dikepalai dengan warna biru dan langkah yang telah dilalui akan berwarna kuning. Langkah yang terinjak lebih dari satu kali akan mulai menggelap



Gambar 4.4.1

Hasil pencarian menggunakan algoritma pencarian DFS menghasilkan rute L-L-D-D-R-R-R-U-U-D-D-L-L-D-D-L. Pencarian dengan DFS memeriksa sebanyak 22 kotak dan membutuhkan 17 langkah pada maze untuk mendapatkan seluruh harta karun



Gambar 4.4.2

Hasil pencarian menggunakan algoritma pencarian DFS menghasilkan rute L-L-D-D-R-R-D-D-L-R-U-U-R-U-U. Pencarian dengan DFS memeriksa sebanyak 17 kotak dan membutuhkan 16 langkah pada maze untuk mendapatkan seluruh harta karun.

4.5 Analisis

4.5.1 Strategi Pencarian

- a. BFS menggunakan strategi "breadth-first", yaitu mengunjungi simpul secara berurutan dari simpul awal, kemudian simpul yang bertetangga dengan simpul awal, dan seterusnya, hingga menemukan simpul tujuan.
- b. DFS menggunakan strategi "depth-first", yaitu mengunjungi simpul secara berurutan dari simpul awal hingga simpul yang tidak memiliki simpul anak lagi

(simpul daun), kemudian mundur ke simpul sebelumnya dan mencoba simpul anak lainnya, hingga menemukan simpul tujuan.

4.5.2 Urutan Pengunjungan

- a. BFS mengunjungi simpul secara berurutan menurut jarak dari simpul awal, sehingga simpul yang lebih dekat akan dikunjungi terlebih dahulu.
- b. DFS mengunjungi simpul secara berurutan menurut kedalaman dari simpul awal, sehingga simpul yang lebih dalam akan dikunjungi terlebih dahulu.

Memori yang Digunakan

- a. BFS menggunakan memori yang cukup besar karena harus menyimpan semua simpul yang dikunjungi dalam level yang sama sebelum melanjutkan ke level berikutnya, serta melakukan reset kembali tiap kali node *treasure* ditemui
- b. DFS menggunakan memori yang lebih sedikit karena hanya perlu menyimpan simpul yang sedang aktif saat ini dan kembali ke simpul sebelumnya setelah menjelajahi semua simpul anak.

Kecepatan Pencarian

- a. BFS biasanya lebih lambat daripada DFS karena harus menjelajahi semua simpul dalam satu level sebelum melanjutkan ke level berikutnya.
- b. DFS biasanya lebih cepat daripada BFS karena mencari simpul tujuan secara langsung dan tidak terpengaruh oleh banyaknya simpul pada level yang sama.

Dalam implementasi di tugas besar ini, penggunaan DFS memungkinkan untuk menemukan harta karun lebih cepat daripada BFS karena DFS hanya mencari simpul tujuan secara langsung dan tidak terpengaruh oleh banyaknya simpul pada level yang sama. Selain itu, kode DFS juga menggunakan memori yang lebih sedikit daripada BFS, sehingga lebih efisien

dalam penggunaan memori. Namun, jika maze memiliki struktur yang kompleks dan banyak simpul yang sejajar pada level yang sama, maka penggunaan BFS mungkin akan lebih efektif dan efisien dalam menyelesaikan masalah treasure hunt.

BAB V

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Kami telah berhasil membuat *desktop application* untuk permasalahan *treasure hunt*. Program ini dibuat dengan menggunakan algoritma pencarian BFS (Breadth First Search) dan DFS (Depth First Search) untuk pencarian *treasure* yang ingin ditemukan. Program ini juga dibuat dengan GUI untuk memvisualisasikan langkah - langkah dan hasil pencarian yang dilakukan.

5.2 Saran

Adapun saran yang ditujukan untuk kelompok kami, seperti pengembangan program yang lebih baik dari sisi strategi algoritma, dan modularitas program kami. Selain itu program BFS dan DFS kami belum sempurna, oleh karena itu diperlukan eksplorasi dan pengembangan lebih. GUI yang kami implementasi juga belum berjalan dengan sempurna, diperlukan pengembangan lebih agar GUI bisa berjalan lebih baik dan optimal.

5.3 Refleksi

Selain itu ada refleksi yang ditujukan untuk kelompok kami. Kami belajar banyak hal yang baru dalam pelajaran algoritma BFS dan DFS, dan implementasi GUI. Kami sempat kesulitan dalam mengerjakan ketiga hal tersebut, namun pada akhirnya kami bisa menyelesaikan semua hal tersebut dengan baik.

5.4 Tanggapan

Tanggapan kami terkait tugas besar ini adalah menurut kami tugas besar ini beban kerjanya lebih berat dalam pembuatan GUI tidak dalam strategi algoritma.

DAFTAR PUSTAKA

informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf

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

learn.microsoft.com/en-us/dotnet/csharp/

learn.microsoft.com/en-us/dotnet/maui/?WT.mc_id=dotnet-35129-website&view=net-maui-7.0

LAMPIRAN

Pranala *repository* dapat diakses pada tautan: github.com/AlifioDitya/Tubes2_RealBingChilling

Pranala YouTube dapat diakses pada tautan: youtu.be/oeJhm2XOwT8