

**LAPORAN TUGAS BESAR IF2211 STRATEGI ALGORITMA
PENGAPLIKASIAN ALGORITMA BFS DAN DFS DALAM
IMPLEMENTASI *FOLDER CRAWLING***



Kelompok 50 | K-02

Muhammad Fikri Ranjabi	13520002
Azka Syauqy Irsyad	13520107
M. Syahrul Surya Putra	13520161

INSTITUT TEKNOLOGI BANDUNG

2022

BAB I

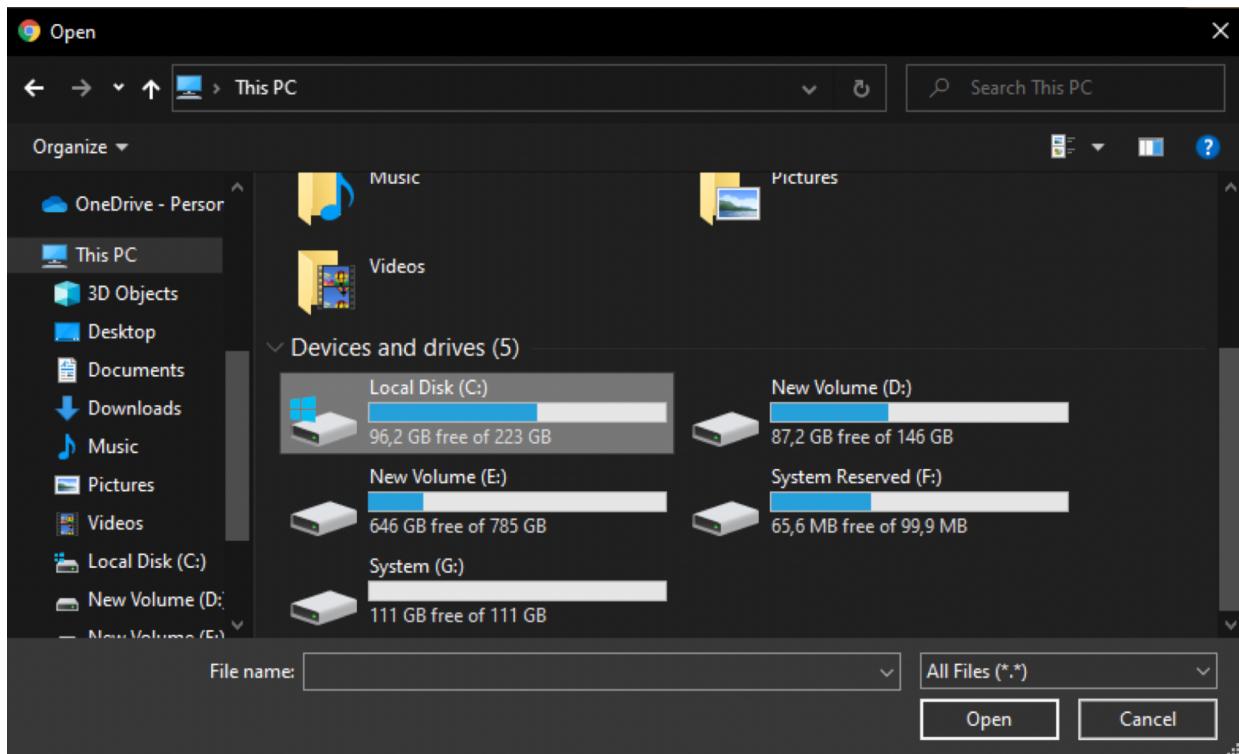
DESKRIPSI TUGAS

Dalam tugas besar ini, Anda akan diminta untuk membangun sebuah aplikasi GUI sederhana yang dapat memodelkan fitur dari *file explorer* pada sistem operasi, yang pada tugas ini disebut dengan *Folder Crawling*. Dengan memanfaatkan algoritma *Breadth First Search* (BFS) dan *Depth First Search* (DFS), Anda dapat menelusuri folder-folder yang ada pada direktori untuk mendapatkan direktori yang Anda inginkan. Anda juga diminta untuk memvisualisasikan hasil dari pencarian *folder* tersebut dalam bentuk pohon.

Selain pohon, Anda diminta juga menampilkan list *path* dari daun-daun yang bersesuaian dengan hasil pencarian. *Path* tersebut diharuskan memiliki *hyperlink* menuju folder *parent* dari file yang dicari, agar file langsung dapat diakses melalui *browser* atau *file explorer*. Contoh hal-hal yang dimaksud akan dijelaskan di bawah ini.

Contoh *input* dan *output* program:

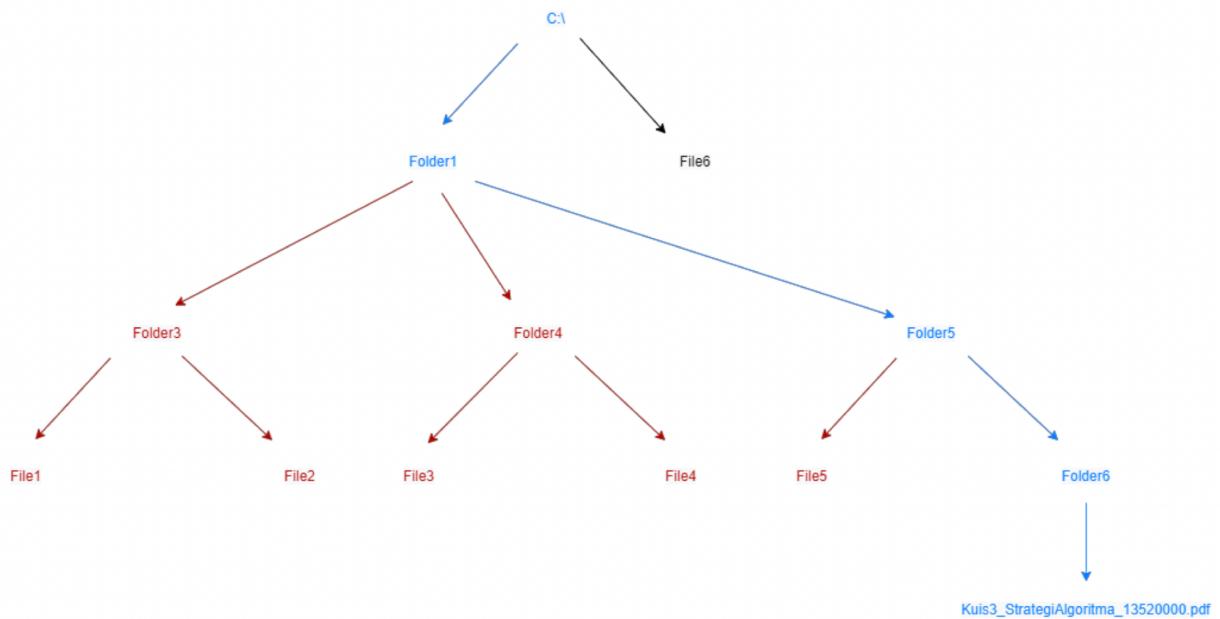
Contoh masukan aplikasi:





Gambar 1 Contoh *input* program

Contoh output aplikasi:



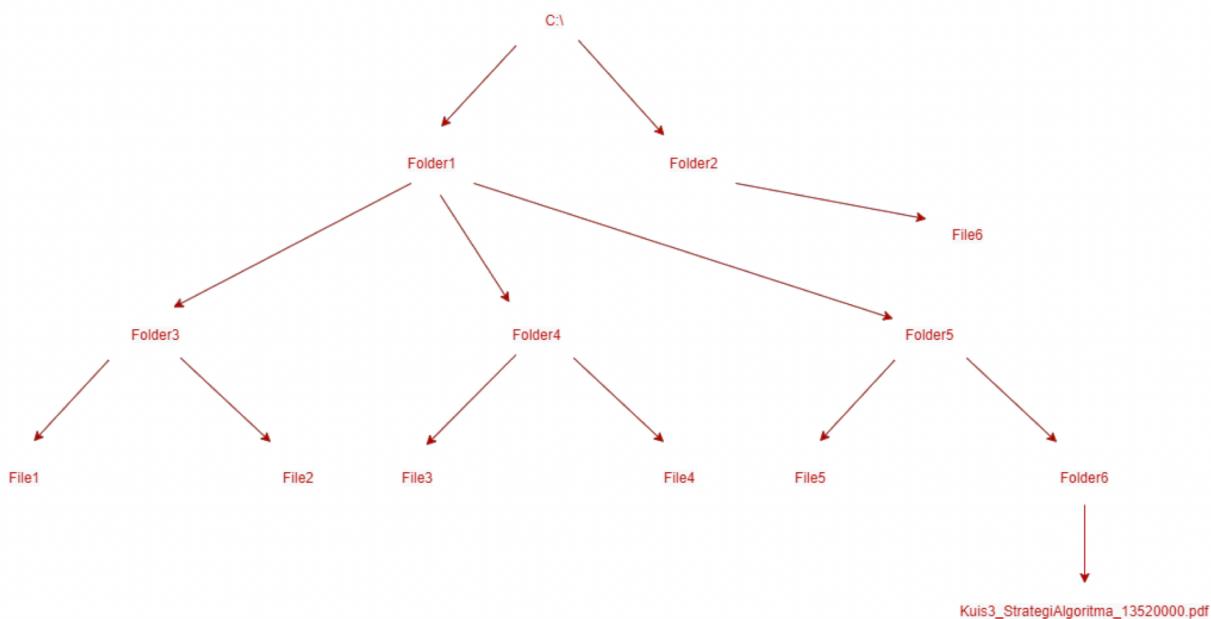
List Path:

1. C:\Folder1\Folder5\Folder6\Kuis3_StrategiAlgoritma_13520000.pdf

Gambar 2 Contoh *output* program

Misalnya pengguna ingin mengetahui langkah *folder crawling* untuk menemukan file Kuis3_StrategiAlgoritma_13520000.pdf. Maka, path pencarian DFS adalah sebagai berikut. C:\ → Folder1 → Folder3 → File1 → Folder3 → File2 → Folder3 → Folder1 → Folder4 → File3 → Folder4 → File4 → Folder4 → Folder1 → Folder5 → File5 → Folder5 → Folder6 → Kuis3_StrategiAlgoritma_13520000.pdf.

Pada gambar di atas, rute yang dilewati pada pencarian DFS diwarnai dengan warna merah. Sedangkan, rute untuk menuju tempat file berada diberi warna biru. Rute yang masuk antrian tapi belum diperiksa diberi warna hitam. Anda bebas menentukan warnanya asalkan dibedakan antara ketiga hal tersebut.



List Path:
Tidak ada path yang sesuai.

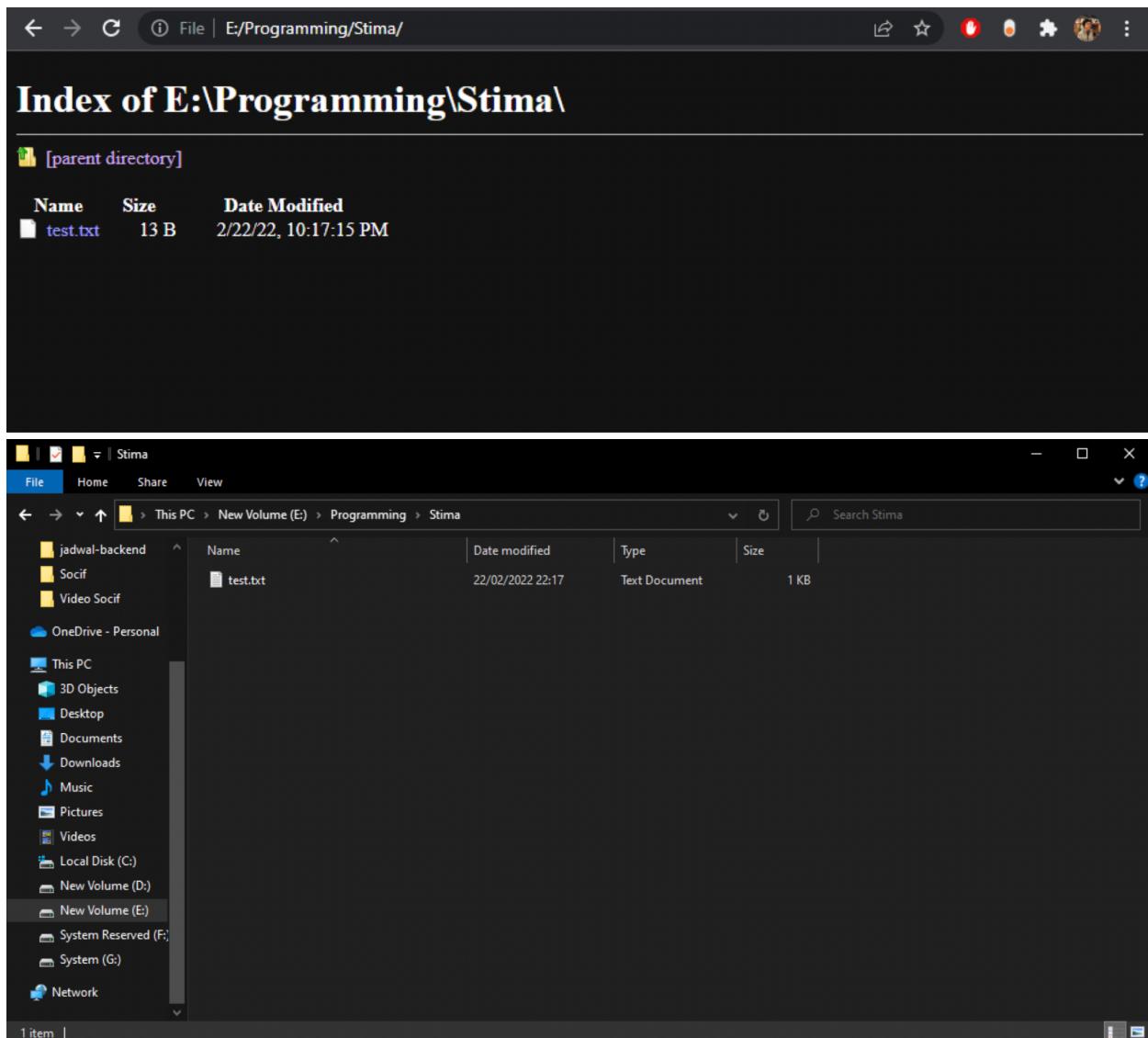
Gambar 3 Contoh *output* program jika file tidak ditemukan

Jika file yang ingin dicari pengguna tidak ada pada direktori file, misalnya saat pengguna mencari Kuis3Probstat.pdf, maka path pencarian DFS adalah sebagai berikut: C:\ → Folder1 → Folder3 → File1 → Folder3 → File2 → Folder3 → Folder1 → Folder4 → File3 → Folder4 → File4 → Folder4 → Folder1 → Folder5 → File5 → Folder5 → Folder6 →

Kuis3_StrategiAlgoritma_13520000.pdf → Folder6 → Folder5 → Folder1 → C:\ → Folder2 → File6.

Pada gambar di atas, semua simpul dan cabang berwarna merah yang menandakan seluruh direktori sudah selesai diperiksa semua namun tidak ada yang mengarah ke tempat file berada.

Contoh *hyperlink* pada path:



Gambar 4 Contoh *hyperlink* ketika diklik

BAB II

LANDASAN TEORI

A. Dasar Teori

Algoritma traversal graf adalah suatu algoritma yang pengaplikasiannya dilakukan dengan mengunjungi simpul dengan cara yang sistematik. Traversal graf ini merupakan suatu algoritma yang dapat digunakan untuk pencarian suatu solusi. Salah dua contoh metode yang ada pada algoritma traversal graf adalah sebagai berikut.

1. *Breadth-First Search* (pencarian melebar)
2. *Depth-First Search* (pencarian mendalam)

1. Algoritma *Breadth-First Search*

Breadth first search, atau pencarian melebar, adalah suatu algoritma yang melakukan proses pencarian secara melebar dengan mengunjungi simpul-simpul yang ada secara *preorder*, yaitu mengunjungi suatu simpul yang di langkah selanjutnya mengunjungi semua simpul yang bertetangga dengan simpul tersebut terlebih dahulu. Simpul-simpul yang belum dikunjungi dan bertetangga dengan simpul yang dikunjungi sebelumnya akan dikunjungi setelahnya, demikian untuk langkah-langkah selanjutnya.

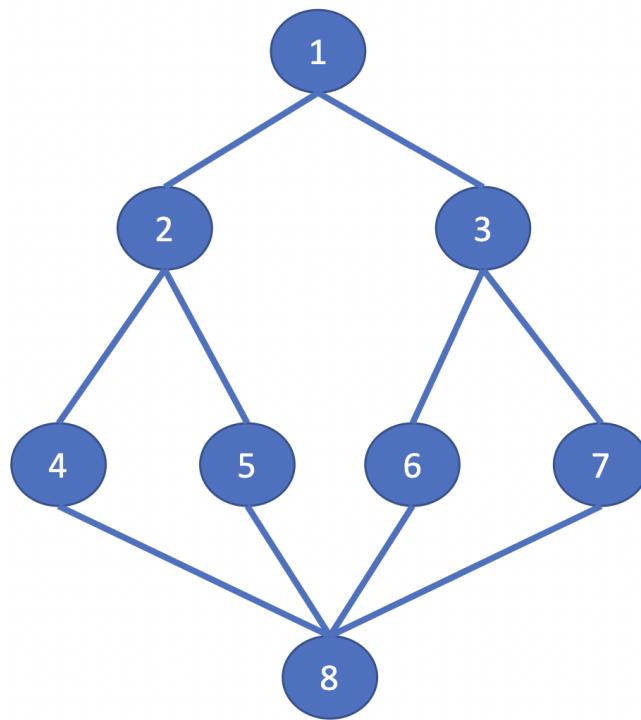
Algoritma BFS ini memerlukan sebuah antrian q yang digunakan untuk menyimpan simpul-simpul yang sudah dikunjungi. Simpul dalam antrian tersebut diperlukan sebagai acuan pengunjungan simpul yang bertetangga dengannya di langkah selanjutnya. Tiap simpul yang telah dikunjungi hanya boleh masuk sebanyak satu kali pada antrian tersebut.

Langkah-langkah dalam pencarian dengan algoritma ini adalah sebagai berikut.

1. Masukkan simpul v pada antrian untuk inisialisasi pencarian pada simpul pertama.
2. Kunjungi simpul pertama yang ada pada antrian, keluarkan simpul tersebut dari antrian dan tandai bahwa simpul tersebut sudah dikunjungi.

3. Masukkan simpul-simpul yang bertetangga yang belum dikunjungi dari simpul yang sedang dikunjungi ke dalam antrian.
4. Kunjungi simpul yang ada pada elemen pertama antrian.
5. Ulangi langkah 3-4 sampai sudah dikunjungi semua simpul yang ada atau mencapai solusi yang sudah ditentukan.

Ilustrasi pengaplikasian algoritma BFS ini adalah sebagai berikut.



Gambar 5 Graf untuk pencarian dengan BFS

Dengan memanfaatkan algoritma BFS, urutan pengunjungan simpul yang terjadi adalah 1-2-3-4-5-6-7-8.

2. Algoritma *Depth First Search*

Depth-first search, atau pencarian mendalam, merupakan suatu algoritma penelusuran graf (atau pohon) yang berkonsep pada kedalaman. Suatu simpul dikunjungi mulai dari *root*, kemudian mengarah kepada salah satu simpul yang merupakan anak dari simpul tersebut.

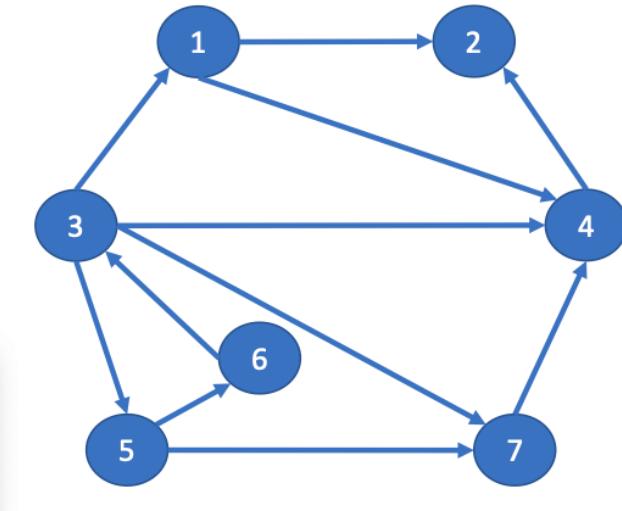
Pencarian dilakukan terus-menerus melalui simpul anak dari simpul yang sedang dikunjungi hingga mencapai pada level kedalaman yang terdalam.

Pada algoritma ini, dikenal yang namanya *backtracking*, yaitu penelusuran kembali ke level kedalaman sebelumnya (simpul dalam) untuk mencari simpul anak lain yang ada pada simpul terkini. Alur pencarian ini terus-menerus dilakukan hingga semua simpul terkunjungi atau suatu solusi telah ditemukan.

Berbeda dengan algoritma BFS, DFS ini memanfaatkan prinsip struktur data *stack*, dimana hal ini dapat dilihat pada pengaplikasian pencarinya yang telah dijabarkan di atas. Adapun langkah-langkah pencarian dalam algoritma ini adalah sebagai berikut.

1. Masukkan simpul v pada *stack* untuk inisialisasi pencarian pada simpul pertama.
2. Kunjungi simpul pertama yang ada pada *stack*, keluarkan simpul tersebut dari *stack* tersebut dan tandai bahwa simpul tersebut sudah dikunjungi.
3. Masukkan dengan *push* simpul anak yang belum dikunjungi dan yang tercapai dari simpul tersebut.
4. Kunjungi simpul pada elemen teratas.
5. Ulangi langkah 3-4 hingga sudah tidak ada lagi simpul anak yang dapat dikunjungi.
6. Jika masih ada simpul yang belum dikunjungi atau solusi belum ditemukan, lakukan proses *backtracking* yang kemudian memasukkan simpul anak lain, dan lakukan kembali langkah 4.
7. Jalankan langkah ke-6 secara berulang hingga semua simpul sudah dikunjungi atau solusi sudah ditemukan.

Ilustrasi pengaplikasian algoritma DFS ini adalah sebagai berikut.



Gambar 6 Graf untuk pencarian dengan DFS

Dengan memanfaatkan algoritma DFS, urutan pengunjungan simpul yang terjadi adalah 1-2-4-3-5-6-7.

B. C# Desktop Application Development

C# merupakan suatu bahasa pemrograman yang mempunyai berbagai fitur yang menarik, tidak kalah dengan bahasa-bahasa pemrograman modern lainnya. Pada bahasa ini, terdapat cara yang paling awam yang memberikan keleluasaan dalam membangun suatu *user interface* berupa *desktop app* dengan menggunakan *windows forms application* dan *toolbox*. *Windows forms* merupakan sekumpulan komponen yang bersifat *reusable* yang mengenkapsulasi fungsionalitas dari *user interface* dan dapat digunakan oleh pengguna pada sistem perangkat berbasis *Windows*. Kontrol pada *form* tersebut merupakan komponen yang berfungsi untuk menampilkan informasi kepada pengguna atau pun menerima *input* dari pengguna.

Untuk membangun proyek dalam C#, perlu menggunakan *Visual Studio IDE*. Di dalam *IDE* tersebut, klik *File*, lalu klik *New Project*. Akan ditampilkan *New Project Dialogue Box* dimana kita dapat memilih bahasa yang akan digunakan. Setelah itu, akan ditampilkan daftar pilihan *project types* dan *templates*, dimana di *templates* ini dipilih opsi *Windows Forms*

Application. Selanjutnya, kita dapat mulai menambahkan berbagai kontrol yang diinginkan pada *form* projek kita.

Pada bagian kiri tampilan, terdapat yang namanya *toolbox*. Ada berbagai macam jenis kontrol yang dikelompokkan di dalam *toolbox* tersebut berdasarkan fungsionalitasnya. Untuk menambahkannya, klik tombol seperti tanda panah di tiap grup yang ada untuk melihat daftar *control* yang ada, yang kemudian dapat ditempatkan di dalam tampilan *form* kita dengan *drag and drop*. Untuk melakukan modifikasi-modifikasi pada kontrol yang ada di *form* yang dibuat, dapat dilakukan dengan melakukan *edit* pada *source code* pada file *form* kita dan menambahkan event pada komponen-komponen yang ada.

BAB III

ANALISIS PEMECAHAN MASALAH

1. Langkah-langkah Pemecahan Masalah

1. Identifikasi masalah: dibutuhkan sebuah program *folder crawling* yang dapat mencari suatu file/folder dari nama yang dimasukkan dan akan ada visualisasi *tree* hasil pencarian dengan metode BFS/DFS.
2. Inisialisasi komponen-komponen yang dibutuhkan ke dalam *windows form* seperti komponen *button*, *textbox*, *label*, *linklabel*, dan *panel graph viewer*.
3. Membuat algoritma yang menginisialisasi semua direktori dan subdirektori yang tersedia dari folder akar ke dalam graf pohon (*tree*).
4. Membuat algoritma BFS dan DFS serta opsi “*find all occurance*” dan pewarnaan pada *tree*.
5. Membuat fungsi yang menampilkan daftar direktori yang ditemukan serta menjalankan *File Explorer* yang membuka direktori yang bersesuaian.
6. Menambahkan fitur *visualizer speed* untuk visualisasi secara *real time*.
7. Meningkatkan tampilan program menggunakan *Material Skin*.

2. Proses Mapping Persoalan menjadi Elemen-elemen Algoritma BFS dan DFS

Pada persoalan *folder crawling*, setiap simpul yang ada pada pohon merepresentasikan nama sebuah folder atau file. Kemudian untuk setiap simpul yang menunjukkan file maka simpul tersebut adalah sebuah daun. Terdapat sebuah fungsi yang digunakan untuk mendapatkan nama folder atau file yang ada pada *path directory*. Misalkan pada path "C:\Users\FikriRanjabi\Desktop\Kuliah Semester 4\IF2211 Strategi Algoritma\Rangkuman Stima.docx" maka didapatkan "Rangkuman Stima.docx" sebagai nama file yang akan menjadi nama simpul dan berlaku juga untuk *path folder*.

Untuk setiap simpul, *parent* simpul adalah *parent path* dan dengan itu maka *child* simpul adalah *child path*. Akar dari pohon dipilih dari *interface* yang disediakan program.

3. Contoh Ilustrasi Kasus

Misalkan pengguna aplikasi pencarian file atau folder yang telah dibuat ingin mencari suatu folder bernama “Folder7”. Untuk menginisiasi pencarian, perlu terlebih dahulu memilih suatu *start directory* dimana berperan sebagai akar pencarian folder

tersebut, dimana dalam hal ini dipilih “C1”. Pengguna selanjutnya dapat menuliskan nama folder atau file yang ingin dicari, dalam hal ini adalah “Folder7”, dan juga memilih metode pencarian, apakah menggunakan BFS atau DFS. Selain itu, pengguna juga dapat memilih kecepatan visualisasi pewarnaan dari *tree* yang nantinya akan ditunjukkan. Ketika ditekan tombol *search*, visualisasi *tree* akan ditampilkan dan pewarnaan *node* juga akan berjalan. *Node* yang memberikan hasil ditemukannya folder tersebut akan berubah warna menjadi biru mulai dari akar hingga jalur ke daun tersebut, dari yang sebelumnya berwarna merah. Ketika ditemukan folder yang dicari, maka aplikasi akan memberikan *hyperlink* yang jika ditekan dapat membawa pengguna ke *directory* sesuai hasil pencarian. Aplikasi juga akan memberikan lama waktu eksekusi pencarian folder tersebut.

Selain itu, misalkan kita mencari suatu file bernama “stima.txt” dengan *starting directory* yang sama dengan ilustrasi kasus sebelumnya. Dengan alur-alur yang sama selanjutnya, program akan mencari file tersebut dan ternyata setelah dievaluasi seluruh *node* yang ada, file tersebut tidak dapat ditemukan. Aplikasi akan menampilkan visualisasi *tree* dengan keseluruhan *node* berwarna merah dan *hyperlink* tidak akan dapat menyimpan suatu *anchor* ke file tersebut serta bertuliskan *None*. Untuk lebih jelasnya, kedua ilustrasi kasus tersebut dapat dilihat pada bagian hasil implementasi yang dicantumkan pada bagian bab selanjutnya.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

1. Implementasi Program

```
procedure initVisited(input node: Node, input/output visited:  
Dictionary of string)  
{ F.S. add all file and folder directory node status to visited  
dictionary and set it's value to false }
```

ALGORITMA

```
foreach (Node temp in node.Children)  
    visited.Add(temp.path, false)  
    initVisited(temp, visited)
```

```
function DFS(input/output foundPath: List of string, input  
allOccurrence: boolean, input searchVal: string, input/output visited:  
Dictionary of boolean, input/output tree: node, input graph: Graph)  
-> Task  
{ Menjalankan pewarnaan DFS pada graf }
```

KAMUS LOKAL

```
tempSearchVal : string
```

ALGORITMA

```
if (not(allOccurrence) and GlobalVar.found) then  
    ->  
  
{ using regex to find searchVal followed by whitespace at the end of  
the word }  
tempSearchVal <- @"^" + searchVal + @"\s*\b"  
  
{ set visited node color to red }  
graph.FindNode(tree.Name).Attr.Color <-  
Microsoft.Msagl.Drawing.Color.Red  
  
if (tree.prevPath != null) then  
    GlobalVar.edges[tree.prevPath.path + tree.path].Attr.Color <-
```

```

MsaglDraw.Color.Red
Form1.updateGraph()
await Task.Delay(delay)

visited[tree.path] <- true

{ set found node and it's parent color to blue }
if (Regex.IsMatch(tree.Name, tempSearchVal)) then
    graph.FindNode(tree.Name).Attr.Color <-
Microsoft.Msagl.Drawing.Color.Blue
colorPath(tree.prevPath, tree, "blue", graph)
foundPath.Add(tree.path)
Form1.updateGraph()
await Task.Delay(delay)
if (not(allOccurence))
    GlobalVar.found <- true
->

foreach (Node temp in tree.Children)
if (not(visited[temp.path])) then
    await DFS(foundPath, allOccurence, searchVal, visited, temp,
graph)
->

```

```

function BFS(input/output path: string, input startNode: Node, input
searchVal: string, input/output graph: Graph, input allOccurence:
boolean) -> Task

```

KAMUS LOKAL

queue : Queue of Node

ALGORITMA

```

bool found <- false
searchVal <- @"^" + searchVal + @"\s*\b"

queue <- new Queue<Node>()
GlobalVar.visited[startNode.path] <- true

```

```

{ set found node and it's parent color to blue }
if (Regex.IsMatch(startNode.Name, searchVal)) then
    graph.FindNode(startNode.Name).Attr.Color <-
Microsoft.Msagl.Drawing.Color.Blue
    Form1.updateGraph()
    await Task.Delay(delay)
    found <- true
    GlobalVar.foundPath.Add(startNode.path)
    if (not(allOccurence)) then
        ->
else
    graph.FindNode(startNode.Name).Attr.Color <-
Microsoft.Msagl.Drawing.Color.Red
    Form1.updateGraph()
    await Task.Delay(delay)

queue.Enqueue(startNode)

{ perform bfs on queue }
while ((queue.Count > 0) and not(found)) do
    Node curNode <- queue.Dequeue()
    foreach (Node temp in curNode.Children)
        if (not(GlobalVar.visited[temp.path]))
            GlobalVar.visited[temp.path] <- true
            queue.Enqueue(temp)

            GlobalVar.edges[temp.prevPath.path + temp.path].Attr.Color
<- MsaglDraw.Color.Red
            Form1.updateGraph()
            await Task.Delay(delay)

{ set found node and it's parent color to blue }
if (Regex.Match(temp.Name, searchVal).Success)
    colorPath(curNode, temp, "blue", graph)
    graph.FindNode(temp.Name).Attr.Color <-
Microsoft.Msagl.Drawing.Color.Blue
    Form1.updateGraph()
    await Task.Delay(delay)
    GlobalVar.foundPath.Add(temp.path)

```

```

        if (not(allOccurrence))
            found <- true
            break
        else
            graph.FindNode(temp.Name).Attr.Color <-
Microsoft.Msagl.Drawing.Color.Red
            Form1.updateGraph()
            await Task.Delay(delay)

```

```

procedure colorPath(input source: Node, input target: Node, input
color: string, input/output graph: Graph)
{ F.S. fill the edge color between source and target node to it's
parent }

```

```

ALGORITMA
if (target.prevPath != null) then
    if (color <- "red") then
        GlobalVar.edges[source.path + target.path].Attr.Color <-
MsaglDraw.Color.Red
        graph.FindNode(target.Name).Attr.Color <-
Microsoft.Msagl.Drawing.Color.Red
        graph.FindNode(source.Name).Attr.Color <-
Microsoft.Msagl.Drawing.Color.Red
    else if (color <- "blue") then
        GlobalVar.edges[source.path + target.path].Attr.Color <-
MsaglDraw.Color.Blue
        graph.FindNode(target.Name).Attr.Color <-
Microsoft.Msagl.Drawing.Color.Blue
        graph.FindNode(source.Name).Attr.Color <-
Microsoft.Msagl.Drawing.Color.Blue
    colorPath(source.prevPath, target.prevPath, color, graph)

```

```

procedure findDir(input path: string, input/output graph: Graph,
input tree: Node)
{ F.S. draw all file and folder directory to graph }

```

KAMUS LOKAL

```

root : string
dirs : list of string
files : list of string
parentFolderName : string

ALGORITMA
string root <- tree.Name

{ enumerate all possible folder and file path }
List<string> dirs <- new
List<string>(Directory.EnumerateDirectories(path))
List<string> files <- new
List<string>(Directory.EnumerateFiles(path))

parentFolderName <- ""
if (System.IO.Directory.GetDirectories(path).Length > 0) then
    foreach (var dir in dirs)
        parentFolderName <-
dir.Substring(dir.LastIndexOf(Path.DirectorySeparatorChar) + 1)

        if (graph.FindNode(parentFolderName) != null) then
            GlobalVar.folderAddon <- GlobalVar.folderAddon + " "
            parentFolderName <- parentFolderName +
GlobalVar.folderAddon
            Node temp <- new Node(parentFolderName, tree, dir, new
List<Node>())
            tree.Children.Add(temp)

            Edge tempEdge <- graph.AddEdge(root, parentFolderName)
            Form1.updateGraph()
            GlobalVar.edges.Add(path + dir, tempEdge)
            await findDir(temp.path, graph, temp)

foreach (var file in files)
    GlobalVar.parentFileName <-
file.Substring(file.LastIndexOf(Path.DirectorySeparatorChar) + 1)
    if (graph.FindNode(GlobalVar.parentFileName) != null) then
        GlobalVar.fileAddon <- GlobalVar.fileAddon + " "
        GlobalVar.parentFileName <- GlobalVar.parentFileName +

```

```
GlobalVar.addon
```

```
    Node temp <- new Node(GlobalVar.parentFileName, tree, file, new  
List<Node>())  
  
    tree.Children.Add(temp)  
    Edge tempEdge <- graph.AddEdge(root, GlobalVar.parentFileName)  
    Form1.updateGraph()  
    GlobalVar.edges.Add(path + file, tempEdge)
```

```
procedure Launch()
```

```
{ F.S. Execute findDir procedure and perform BFS/DFS searching }
```

```
KAMUS LOKAL
```

```
path : string  
tree : Node
```

```
ALGORITMA
```

```
string path <- GlobalVar.selectedPath
```

```
{ create tree }
```

```
Node tree <- new  
Node(path.Substring(path.LastIndexOf(Path.DirectorySeparatorChar) +  
1), null, path, new List<Node>())
```

```
findDir(tree.path, GlobalVar.graph, tree)
```

```
{ add root to visited dictionary }
```

```
GlobalVar.visited.Add(tree.path, false)
```

```
initVisited(tree, GlobalVar.visited)
```

```
if (GlobalVar.method = "DFS") then
```

```
    await DFS(GlobalVar.foundPath, GlobalVar.allOccurrence,
```

```
GlobalVar.searchVal, GlobalVar.visited, tree, GlobalVar.graph)
```

```
else if (GlobalVar.method = "BFS") then
```

```
    await BFS(GlobalVar.selectedPath, tree, GlobalVar.searchVal,
```

```
GlobalVar.graph, GlobalVar.allOccurrence)
```

```

procedure (searchBtn_Click)
{ Menjalankan BFS/DFS ketika tombol "Search" diklik }

ALGORITMA
if (GlobalVar.isFolderChoosen) then
    delay <- int.Parse(Program.form1.trackBar1.Value.ToString())
    GlobalVar.found <- false
    GlobalVar.graph <- new Microsoft.Msagl.Drawing.Graph("graph")
    GlobalVar.foundPath.Clear()

    string searchingPath <- materialLabel1.Text + "\\...\\"
    materialSingleLineTextField1.Text
    MaterialLabel7.Text <- searchingPath

    { assign file/folder name to search value }
    GlobalVar.searchVal <-
    Regex.Escape(materialSingleLineTextField1.Text)
    GlobalVar.edges.Clear()
    GlobalVar.visited.Clear()

    var watch <- new System.Diagnostics.Stopwatch()
    watch.Start()
    var task <- GraphViewer.Launch()
    await Task.WhenAll(task)
    watch.Stop()
    materialLabel9.Text <- (watch.ElapsedMilliseconds).ToString() +
ms"

    initializedFoundPath() { add found path to linklabel }

else
    MessageBox.Show("Choose Folder First!")

```

```

{ user defined node datatype }
type Node: <Name: string,
            prevPath: Node,
            path: string,

```

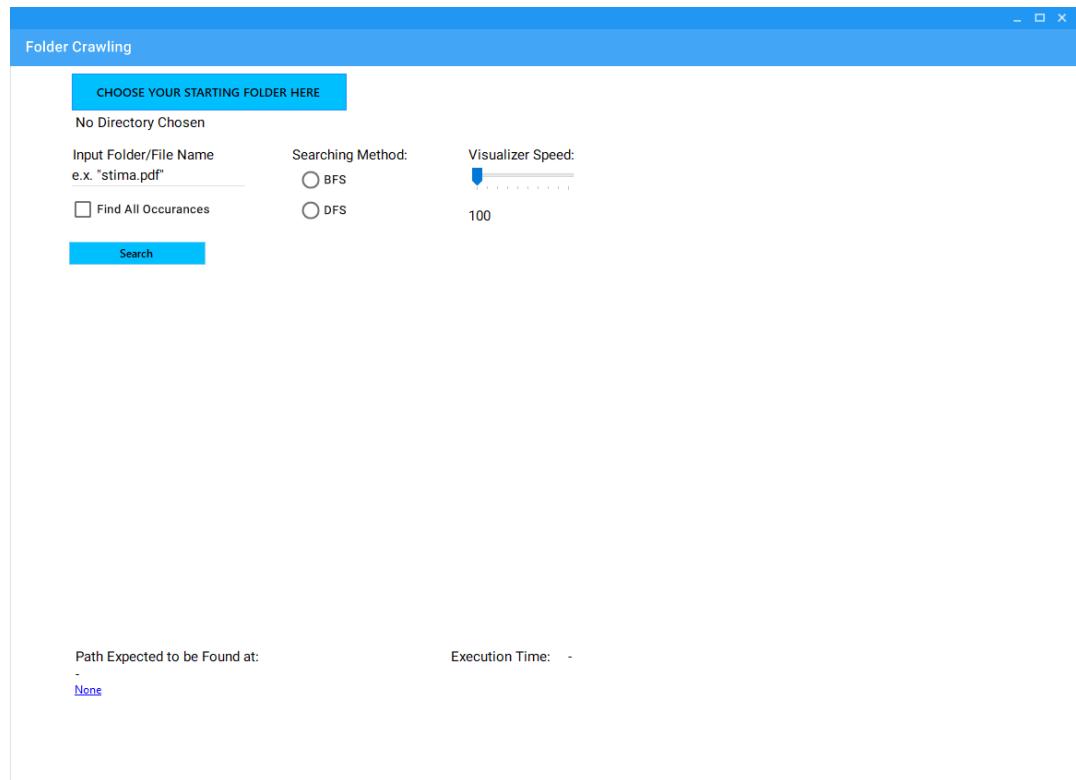
2. Spesifikasi Program dan Struktur Data

Program ditulis menggunakan bahasa pemrograman C# dengan IDE Microsoft Visual Studio. *Project* dibuat dengan memilih *template Windows Forms App (.NET Framework)* yang telah disediakan Microsoft Visual Studio. Pembuatan dan penampilan graf memanfaatkan *library* MSAGL. Tampilan program menggunakan library Material Skin. Berikut adalah beberapa struktur data yang digunakan.

Nama Variable	Struktur Data
Edge	Edge
edges	Dictionary of Edge
visited	Dictionary of Boolean
foundPath	List of String

3. Penggunaan Program

a. Interface Program



Gambar 7 Tampilan *interface* program

b. Fitur-fitur Program

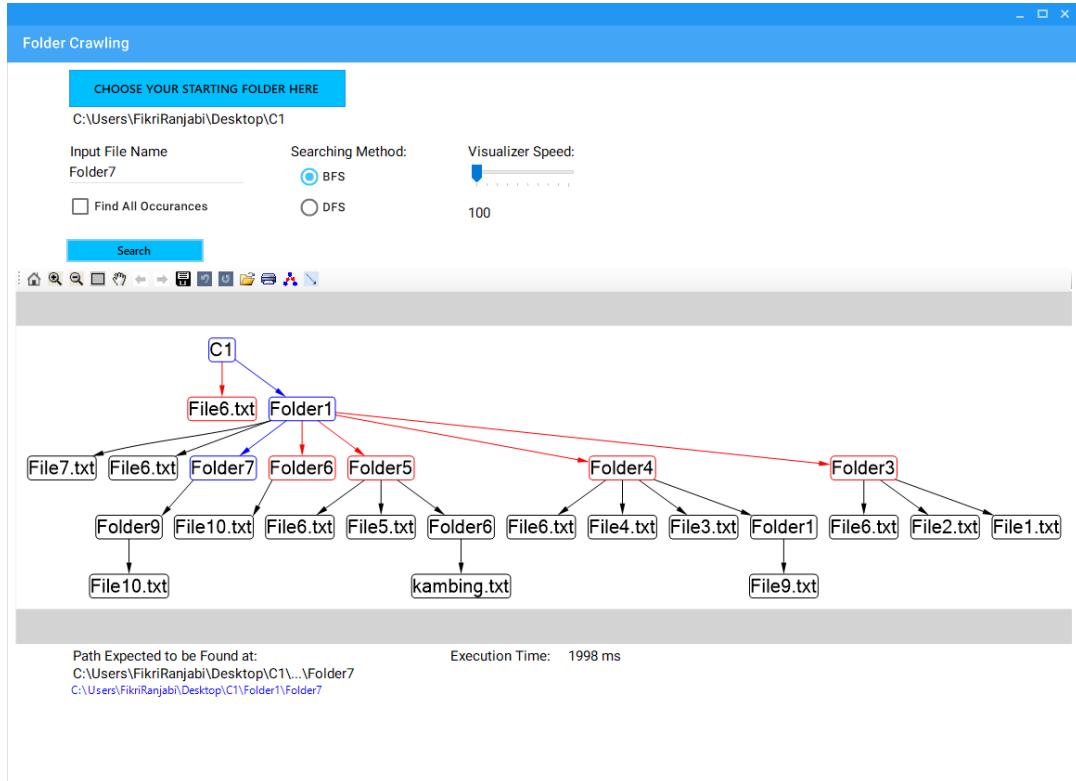
Nama Fitur	Keterangan
Choose Starting Directory	Pada fitur ini dapat dipilih sebuah folder sebagai direktori awal pencarian dan akan menjadi akar dari pohon.
Input Folder/File Name	Diisi dengan nama file/folder yang ingin dicari.
Find All Occurrence	Jika ingin mencari semua kemunculan folder/file dengan nama yang sama, maka opsi ini dapat dipilih.
Searching Method	Dapat dipilih metode pencarian yang ingin digunakan yaitu Breadth First

	Search (BFS) atau Depth First Search (DFS).
Visualizer Speed	Kecepatan pencarian pada visualisasi graf dapat diatur dengan menggeser slider ini.
Path Founded	Setelah pencarian selesai, akan ditampilkan path-path yang ditemukan dan path tersebut dapat diklik yang nantinya menampilkan aplikasi Windows Explorer di path terkait.
Time Execution	Setelah pencarian selesai, akan ditampilkan lama waktu eksekusi pencarian program
Graph Visualizer	Visualisasi akan ditampilkan pada panel bagian bawah program.
Material Skin	Tampilan program.

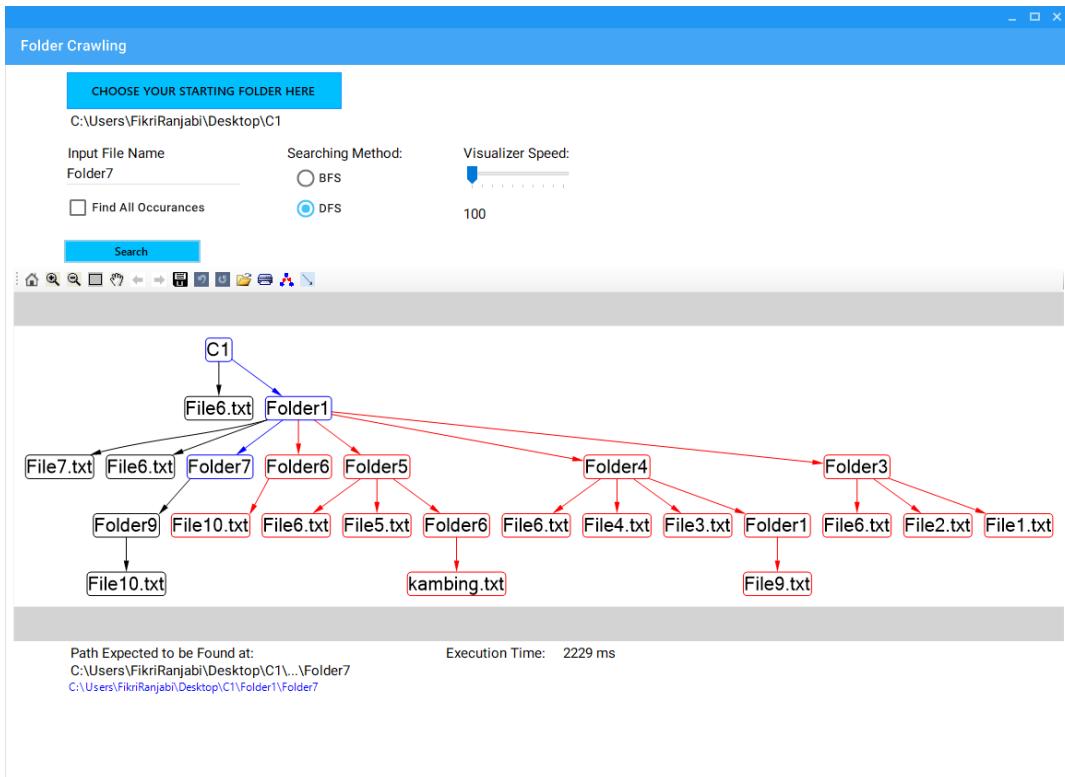
c. Alur Penggunaan Program

Setelah program dibuka, maka pilih sebuah folder sebagai direktori awal pencarian dengan menekan tombol “Choose Folder”. Kemudian masukkan nama folder/file yang ingin dicari pada form di bagian Input Folder/File Name. Jika ingin mencari semua folder/file dengan nama yang sama, maka centang opsi “Find All Occurrence”. Setelah itu pilih metode pencarian yang digunakan (BFS/DFS). Atur kecepatan visualisasi pencarian. Kemudian tekan tombol “Search”. Visualiasi graf akan muncul dan kemudian path folder/file yang ditemukan akan berada pada bagian “Path Founded”. Path yang ada dapat diklik yang nantinya akan terbuka Windows Explorer pada path yang bersangkutan.

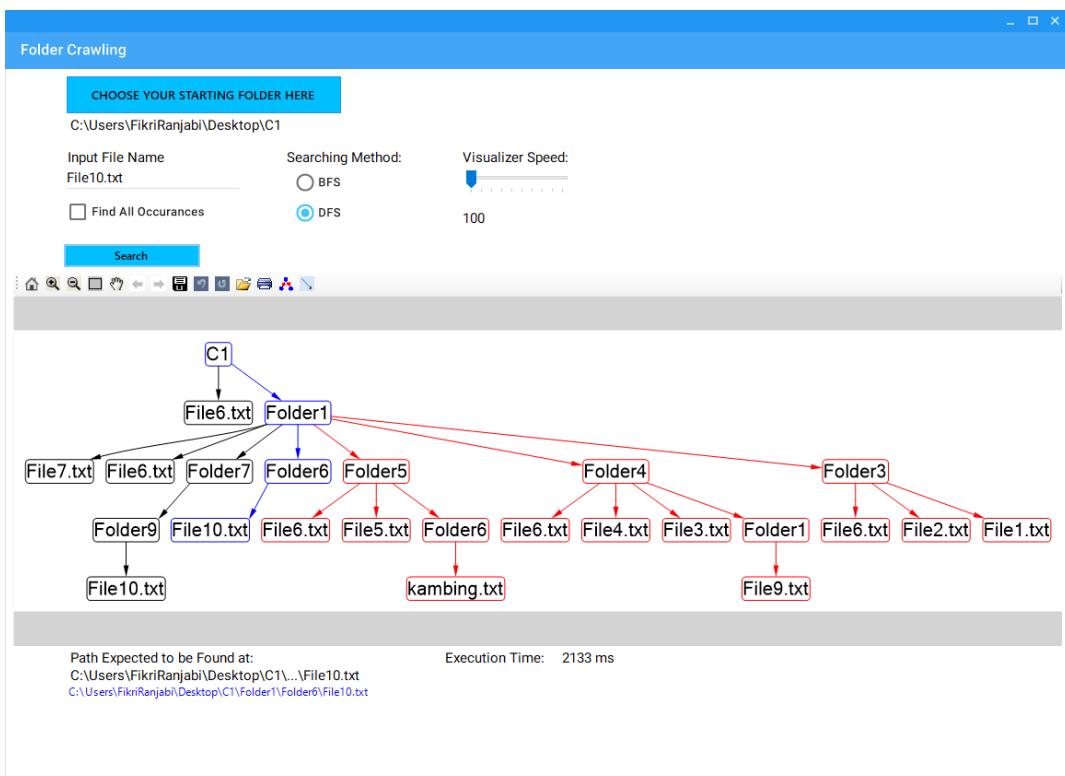
4. Hasil Pengujian Program



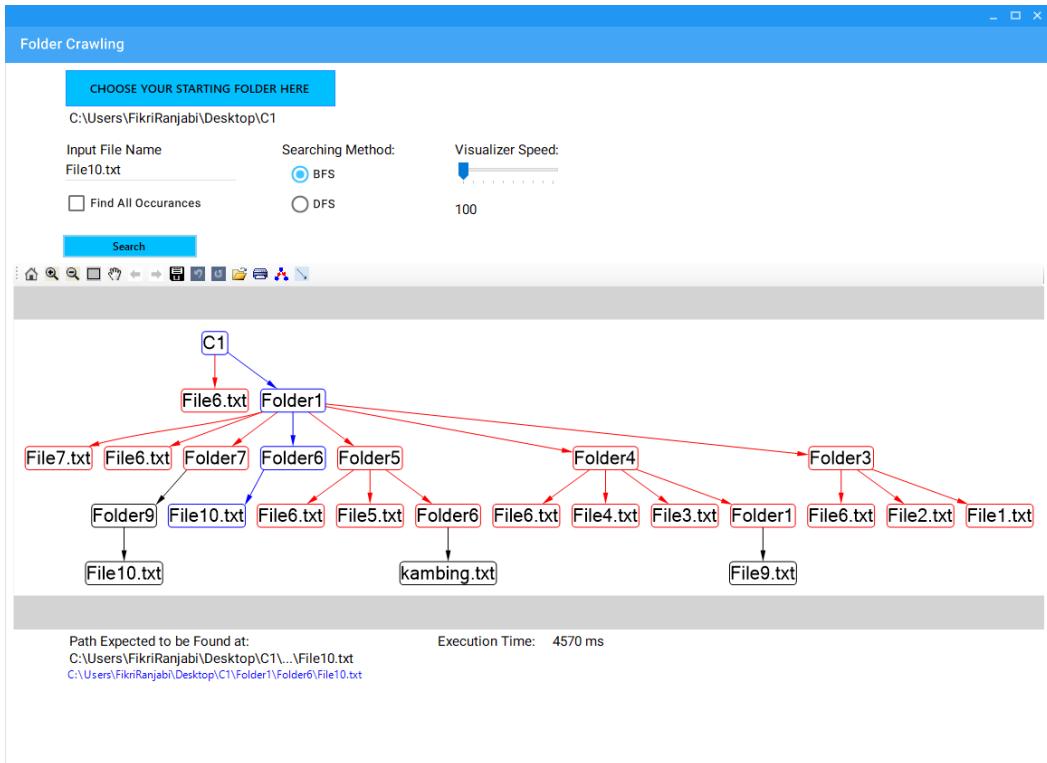
Gambar 8 Pencarian Folder7 dengan BFS



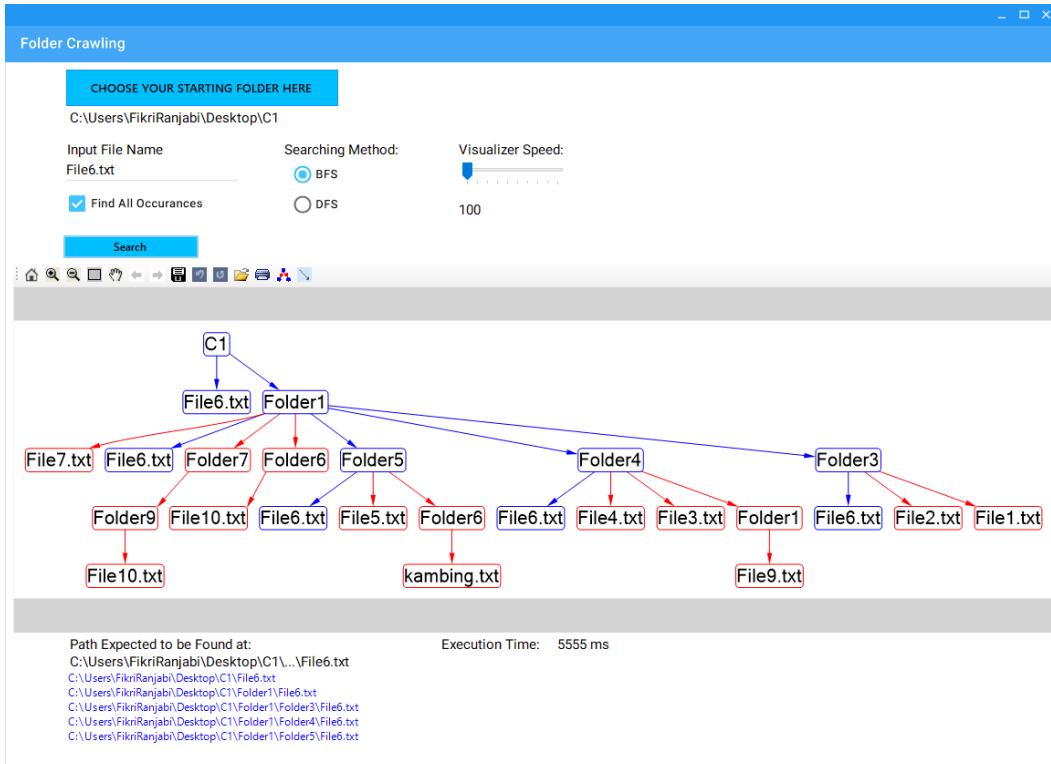
Gambar 9 Pencarian Folder7 dengan DFS



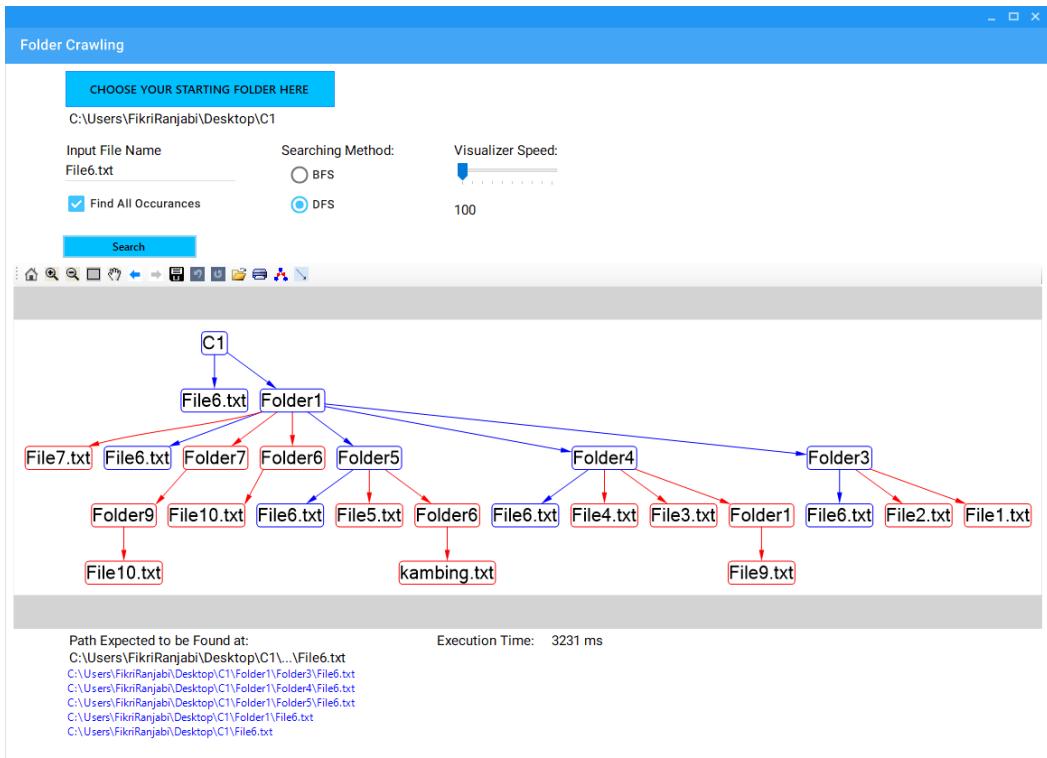
Gambar 10 Pencarian File10.txt dengan DFS



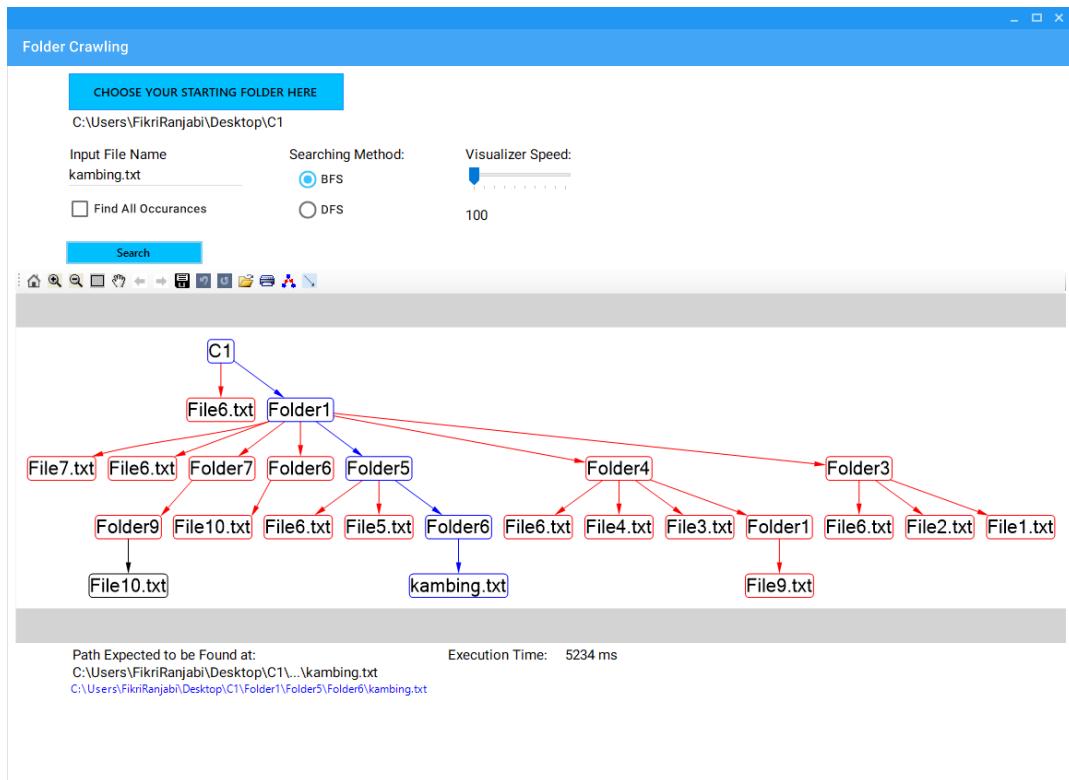
Gambar 11 Pencarian File10.txt dengan BFS



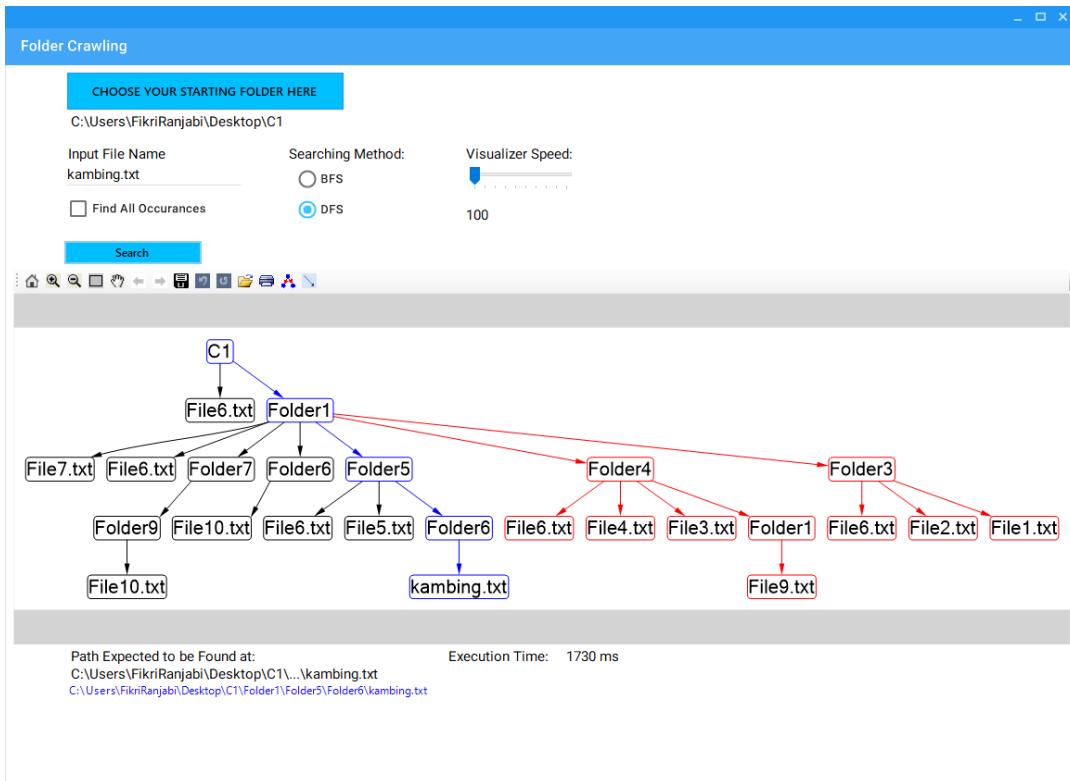
Gambar 12 Pencarian seluruh File6.txt dengan BFS



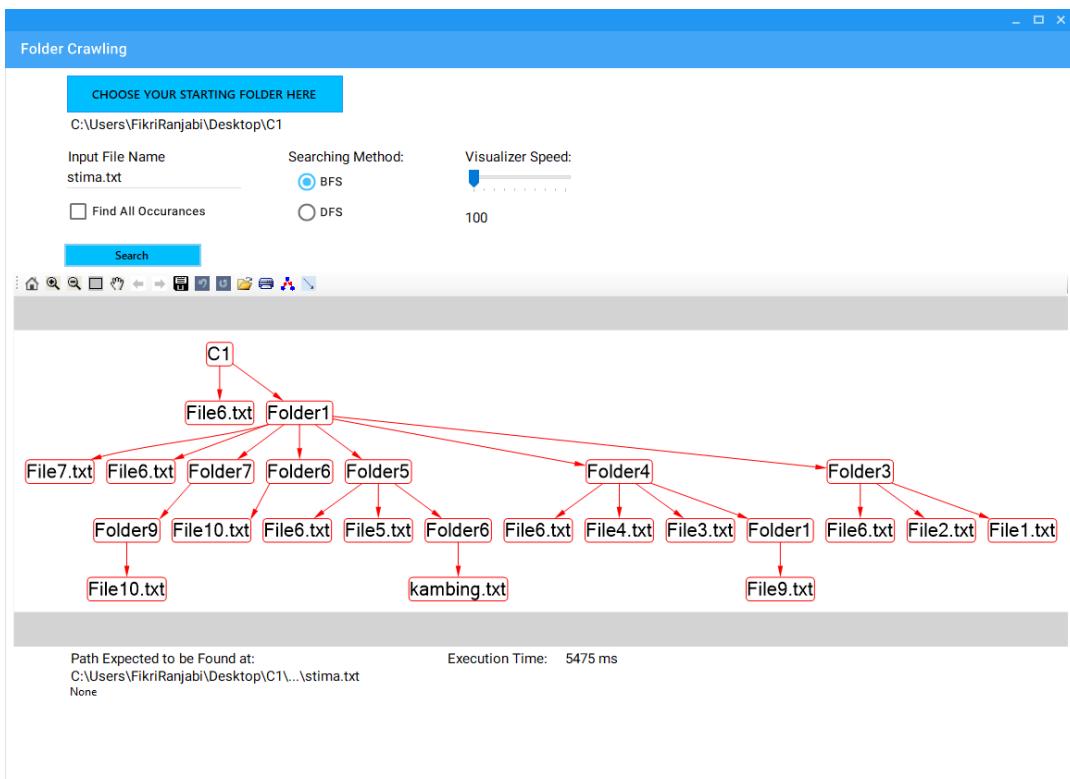
Gambar 13 Pencarian seluruh File6.txt dengan DFS



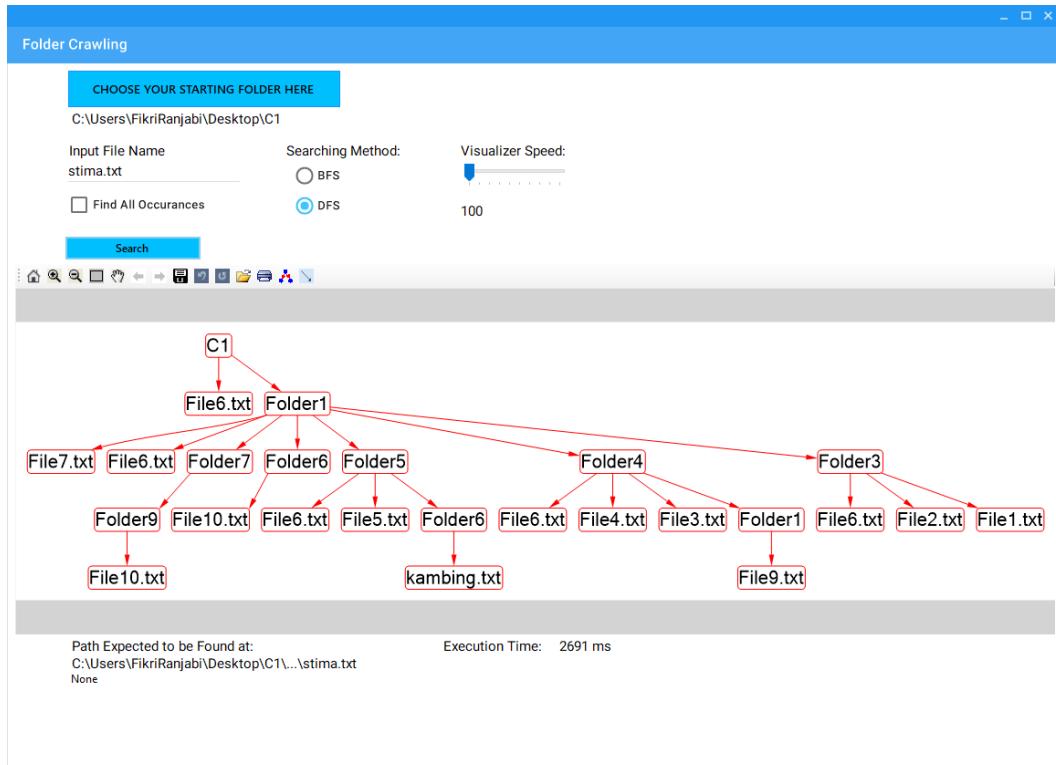
Gambar 14 Pencarian kambing.txt dengan BFS



Gambar 15 Pencarian kambing.txt dengan DFS



Gambar 16 Pencarian stima.txt dengan BFS



Gambar 17 Pencarian stima.txt dengan DFS

5. Analisis Desain Solusi Algoritma BFS dan DFS

Dilihat pada percobaan pencarian Folder7 yang telah dicantumkan di atas, dimana letak kedalaman simpul tersebut pada graf hasil visualisasi berada pada 3, dapat diperhatikan bahwa hasil waktu eksekusi pencarian dengan metode BFS maupun DFS tidak jauh berbeda, walaupun jika dilihat pada tampilan program masih sedikit lebih cepat dengan menggunakan metode BFS. Untuk perbandingan jumlah *step* yang terjadi, algoritma BFS memakan *step* yang lebih sedikit, yaitu sebesar 8, daripada DFS yang memakan sebesar 20. Pada percobaan pencarian File10.txt dengan kedalaman 4, dapat dilihat bahwa perbedaan hasil waktu eksekusi sebesar hampir dua kali lipatnya, dimana waktu eksekusi metode DFS yang lebih cepat dan jumlah *step* yang lebih sedikit dibanding metode BFS. Bahkan, untuk kedalaman 5 dengan pencarian kambing.txt, beda waktu eksekusi terjadi lebih dari dua kali lipat dengan kecepatan BFS yang jauh lebih

lambat daripada DFS. Hal yang serupa juga terjadi pada pencarian File6.txt dengan mencari semua kemungkinan penemuan yang terjadi, dimana metode pencarian DFS masih lebih cepat dibandingkan metode pencarian BFS.

Jika melihat percobaan pencarian stima pada gambar di atas, dimana pencarian tidak menemukan solusi, dapat kita lihat bahwa waktu eksekusi pencarian dengan BFS dan DFS menunjukkan waktu yang berbeda walaupun semua simpul dikunjungi secara keseluruhan yang menunjukkan bahwa *step* yang dijalani di dua algoritma tersebut sama. Menurut hasil analisis yang dilakukan terhadap program yang telah dibuat, perbedaan waktu tersebut dapat disebabkan oleh beberapa hal. Misalnya adalah banyaknya *background task* yang terjadi secara bersamaan sehingga *measuring time* yang terjadi berbeda di tiap eksekusi programnya. Selain itu, salah satu hal menonjol adalah desain algoritma DFS dan BFS yang diaplikasikan. Pada implementasi metode DFS, pencarian dilakukan dengan metode rekursif, sedangkan metode implementasi BFS menggunakan struktur data *queue*. Hal ini dapat membuat perbedaan waktu eksekusi ketika program di *compile* oleh mesin dan di *execute*.

BAB V

KESIMPULAN DAN SARAN

Algoritma *depth-first search* (DFS) dan *breadth-first search* (BFS) merupakan satus algoritma graf traversal yang berguna untuk penelusuran pencarian solusi. Kedua algoritma ini dapat diterapkan dalam program *folder crawling*, yaitu pencarian suatu folder atau file dari suatu *starting directory*. Dengan memanfaatkan juga bahasa pemrograman C# sebagai bahasa dalam pembangunan suatu aplikasi berbasis *desktop*, algoritma DFS dan BFS berhasil melakukan eksplorasi pencarian, dimana pada program ini digambarkan sebagai suatu pohon, hingga ditemukan suatu simpul yang merupakan file atau folder yang dicari atau sampai seluruh simpul telah dikunjungi. Tentu pencarian di tiap algoritmanya disesuaikan dengan *logic* dari BFS atau pun DFS, dimana BFS bekerja seperti pada struktur data *queue* sedangkan DFS bekerja seperti struktur data *stack*.

Saran untuk implementasi tugas-tugas besar selanjutnya, akan lebih baik apabila implementasi untuk program tidak bergantung pada *platform/operating system* tertentu. Seperti misalnya pada tugas besar ini yang menggunakan *windows control forms* yang menyebabkan hanya bisa digunakan pada komputer berbasis *windows*. Untuk mahasiswa yang menggunakan jenis OS yang berbeda menjadi tidak leluasa mengerjakannya karena perlu bertemu dengan anggota kelompok lainnya yang memenuhi kriteria tersebut sehingga cukup menghambat dari segi efektivitas waktu, tenaga, dan biaya.

DAFTAR PUSTAKA

[1]

https://cdn-edunex.itb.ac.id/38015-Algorithm-Strategies-Parallel-Class/85259-BFS-dan-DFS/1646201812962_Tugas-Besar-2-IF2211-Strategi-Algoritma-2022.pdf

[2] <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>

[3] http://csharp.net-informations.com/gui/gui_tutorial.htm

LAMPIRAN

Repository Github:

<https://github.com/ranjabi/Folder-Crawling-BFS-DFS>

Video Demonstrasi:

https://youtu.be/-E5_KsOAIvg