

**LAPORAN TUGAS BESAR 2 IF2211 STRATEGI
ALGORITMA**

SEMESTER II TAHUN 2020/2021

**PENGAPLIKASIAN ALGORITMA BFS DAN DFS
DALAM IMPLEMENTASI *FOLDER CRAWLING***



Disusun oleh:

Dwi Kalam Amal Tauhid	13519210
Daniel Salim	13520008
Kevin Roni	13520114

DAFTAR ISI

BAB 1	3
DESKRIPSI TUGAS	3
1.1 Spesifikasi Program	3
1.2 Spesifikasi GUI	5
BAB 2	7
LANDASAN TEORI	7
2.1 Dasar Teori Secara Umum	7
2.2 Penjelasan singkat mengenai C# desktop application development	8
BAB 3	9
ANALISIS PEMECAHAN MASALAH	9
3.1 Langkah-langkah Pemecahan Masalah	9
3.2 Proses Mapping Persoalan Menjadi Elemen-elemen Algoritma BFS dan DFS.	9
3.3 Contoh Ilustrasi Kasus Lain Yang Berbeda Dengan Contoh Pada Spesifikasi Tugas	11
BAB 4	12
IMPLEMENTASI DAN PENGUJIAN	12
4.1 Implementasi Program	12
4.2 Spesifikasi Program dan Struktur Data	27
4.3 Penjelasan dan Tata Cara Penggunaan Program	27
4.4 Hasil Pengujian	28
4.5 Analisis Desain Solusi Algoritma BFS dan DFS	32
BAB 5	34
KESIMPULAN DAN SARAN	34
5.1 Kesimpulan	34
5.2 Saran	34
DAFTAR PUSTAKA	35
LAMPIRAN	36
Repository Github	36
Video Demonstrasi	36

BAB 1

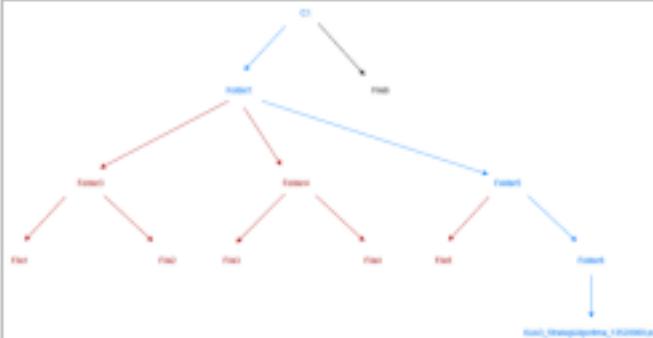
DESKRIPSI TUGAS

Pada tugas besar ini, penulis 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.

1.1 Spesifikasi Program

Aplikasi yang akan dibangun dibuat berbasis GUI. Salah satu contoh tampilan dari aplikasi GUI yang dapat dibangun adalah sebagai berikut.

Folder Crawling	
<p>Input</p> <p>Choose Starting Directory <input type="button" value="Choose Folder..."/> No File Chosen</p> <p>Input File Name <input type="text" value="e.g. 'word.pdf'"/></p> <p><input type="checkbox"/> Find all occurrence</p> <p>Input Metode Pencarian <input type="radio"/> BFS <input checked="" type="radio"/> DFS</p> <p><input type="button" value="Search"/></p>	<p>Output</p> 
<p>Folder Crawling</p> <hr/> <p>Input</p> <p>Choose Starting Directory <input type="button" value="Change Folder..."/> C:/</p> <p>Input File Name <input type="text" value="Kuis3_StrategiAlgoritma_13520000.pdf"/></p> <p><input type="checkbox"/> Find all occurrence</p> <p>Input Metode Pencarian <input type="radio"/> BFS <input checked="" type="radio"/> DFS</p> <p><input type="button" value="Search"/></p>	
<p>Output</p>  <pre> graph TD Root[Root] --> F1[File1] Root --> F2[File2] Root --> F3[File3] F1 --> F1_1[File1.1] F1 --> F1_2[File1.2] F2 --> F2_1[File2.1] F2 --> F2_2[File2.2] F3 --> F3_1[File3.1] F3_1 --> Kuis3[Kuis3_StrategiAlgoritma_13520000.pdf] </pre> <p>Path File : • C:/Folder1/Folder2/Folder3/Kuis3_StrategiAlgoritma_13520000.pdf</p> <p>Time spent: 20.02s</p>	

Gambar 1.1. Contoh tampilan layout dari aplikasi desktop yang dibangun

Program yang dibuat harus memenuhi **spesifikasi wajib** sebagai berikut:

1. Program dibangun dalam bahasa **C#** untuk melakukan penelusuran *Folder Crawling* sehingga diperoleh hasil pencarian file yang diinginkan. Penelusuran harus memanfaatkan algoritma **BFS dan DFS**.
2. Awalnya program menerima sebuah input folder pada direktori yang ada dan nama file yang akan dicari oleh program.
3. Terdapat dua pilihan pencarian, yaitu:
 - i. Mencari 1 file saja
Program akan memberhentikan pencarian ketika sudah menemukan file yang memiliki nama sama persis dengan input nama file.
 - ii. Mencari semua kemunculan file pada folder root
Program akan berhenti ketika sudah memeriksa semua file yang terdapat pada folder root dan program akan menampilkan daftar semua rute file yang memiliki nama sama persis dengan input nama file
4. Program kemudian dapat menampilkan **visualisasi pohon pencarian file** berdasarkan informasi direktori dari folder yang di-input. Pohon hasil pencarian file ini memiliki root adalah folder yang di-input dan setiap daunnya adalah file yang ada di folder root tersebut. Setiap folder/file direpresentasikan sebagai sebuah node atau simpul pada pohon. Cabang pada pohon menggambarkan folder/file yang terdapat di folder *parent*-nya. Visualisasi pohon juga harus disertai dengan **keterangan** node yang sudah diperiksa, node yang sudah masuk antrian tapi belum diperiksa, dan node yang bagian dari rute hasil penemuan. Proses visualisasi ini boleh memanfaatkan pustaka atau kakas yang tersedia.
5. Program juga dapat menyediakan *hyperlink* pada setiap hasil rute yang ditemukan. *Hyperlink* ini akan membuka folder parent dari file yang ditemukan. Folder hasil *hyperlink* dapat dibuka dengan *browser* atau *file explorer*.
6. Mahasiswa **tidak diperkenankan** untuk melihat atau menyalin library lain yang mungkin tersedia bebas terkait dengan pemanfaatan BFS dan DFS. Tapi untuk algoritma lainnya seperti *string matching* dan akses *directory*, diperbolehkan menggunakan library jika ada.

1.2 Spesifikasi GUI

GUI yang dibuat memenuhi spesifikasi sebagai berikut.

1. Program dapat menerima input folder dan query nama file.
2. Program dapat memilih untuk menampilkan satu hasil saja atau menemukan semua file yang memiliki nama file sama persis dengan input query

3. Program dapat memilih algoritma yang digunakan.
4. Program dapat menampilkan pohon hasil pencarian file tersebut dengan memberikan keterangan folder/file yang sudah diperiksa, folder/file yang sudah masuk antrian tapi belum diperiksa, dan rute folder serta file yang merupakan rute hasil pertemuan.
5. **(Bonus)** Program dapat menampilkan progress pembentukan pohon dengan menambahkan node/simpul sesuai dengan pemeriksaan folder/file yang sedang berlangsung.
6. Program dapat menampilkan hasil pencarian berupa rute/path (bisa lebih dari satu jika memilih menemukan semua file) serta durasi waktu algoritma.
7. GUI dapat dibuat **sekreatif** mungkin asalkan memuat 5 (6 jika mengerjakan bonus) spesifikasi di atas.

BAB 2

LANDASAN TEORI

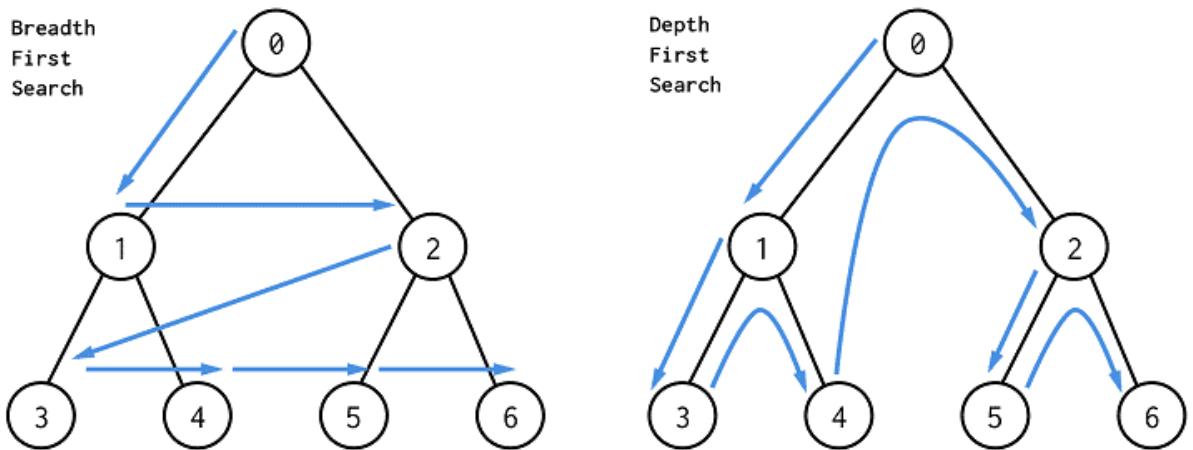
2.1 Dasar Teori Secara Umum

Graf traversal merupakan proses mengunjungi setiap simpul atau *node*. Graf traversal dapat digunakan untuk mencari jalur dari titik asal ke titik tujuan, mencari jalur terpendek antardua simpul, atau menemukan semua jalur yang bisa dilalui dari titik asal ke titik tujuan. Dalam melakukan penjelajahan sebuah graf, khususnya graf terbatas, dapat digunakan algoritma-algoritma seperti *Breadth First Search* (BFS) dan *Depth First Search* (DFS).

Breadth First Search (BFS) merupakan algoritma yang akan mengunjungi setiap simpul graf secara bertahap dari level terawal (setiap simpul *parent*) sebelum secara traversal mengunjungi masing-masing simpul *child*-nya (satu level di bawahnya, dst), hingga semua simpul telah berhasil dikunjungi.

Berbeda dengan BFS, *Depth First Search* (DFS) akan secara traversal mengunjungi simpul *child* yang belum dikunjungi sebelum mengunjungi simpul lainnya yang berada pada satu level yang sama. Jika seluruh simpul *child* sudah dikunjungi atau sudah tidak ada, dilakukan penelusuran terhadap simpul *parent* dari simpul yang terakhir kali dikunjungi untuk dilakukan pengunjungan terhadap simpul *child* yang belum dikunjungi dari simpul *parent* tersebut. Begitu seterusnya hingga semua simpul graf telah dikunjungi.

Berikut adalah ilustrasi algoritma pencarian BFS dan DFS.



Gambar 2.1.1. Ilustrasi Algoritma BFS dan DFS pada Graf

https://miro.medium.com/max/750/0*ZIsIX-f-j7kvxJMW.png

2.2 Penjelasan singkat mengenai C# desktop application development

Dalam mengembangkan aplikasi *desktop* dengan bahasa pemrograman C#, pengembangan dapat dibantu dengan menggunakan *framework* seperti .NET sehingga pemrogram lebih dimudahkan dalam melakukan koding ataupun *debugging*. Tak hanya itu, pengembangan juga dapat semakin dipermudah dengan penggunaan IDE, seperti Visual Studio di sistem operasi Windows atau MonoDevelop di sistem operasi berbasis Linux. Dengan *environment* tersebut, tipe aplikasi yang akan dibangun pun beragam, seperti Windows Forms, WPF, UWP, dll.

BAB 3

ANALISIS PEMECAHAN MASALAH

3.1 Langkah-langkah Pemecahan Masalah

1. Identifikasi masalah: diminta pembuatan sebuah program *folder crawling* yang dapat mencari suatu file dari input pengguna dengan metode yang diinginkan, BFS atau DFS, dan akan dibuat visualisasi rute perjalanan pencarian dengan *tree* dari *root folder* sampai berhasil bertemu dengan file yang dicari.
2. Mempelajari dan memahami: algoritma traversal menggunakan metode BFS & DFS, memahami struktur *directory* penyimpanan file di Windows, bahasa pemrograman C#, Framework .NET, visualisasi graph (MSAgl), dan pembuatan GUI untuk Windows Form.
3. Merancang algoritma pencarian file dengan diketahui root foldernya dengan metode BFS dan DFS.
4. Merancang GUI Program menggunakan Visual Studio.
5. Merealisasikan program.

3.2 Proses Mapping Persoalan Menjadi Elemen-elemen Algoritma BFS dan DFS.

- A. Penelusuran setiap file atau folder dimulai dari *root path* hingga ditemukannya file yang ingin dicari

Dalam pengimplementasian algoritma BFS, elemen utama yang digunakan adalah suatu *method* SearchBFS. Pengunjungan file dan folder yang dieksekusi oleh SearchBFS akan dimulai dari sebuah direktori. Selanjutnya, program akan menerima list berupa daftar files dan direktori yang ada di direktori tersebut dan memasukkannya ke antrian. Prioritas antrian adalah mendahulukan untuk memproses file sebelum direktori. Jika file-file pada suatu direktori tidak ada yang sesuai dengan yang dicari maka pemrosesan berlanjut ke antrian setiap direktori dengan mengakses setiap file dan direktori dari direktori tersebut. Proses yang terjadi adalah sama seperti sebelumnya dalam memproses mulai dari memasukkan antrian memprosesnya. Jika

tidak ditemukan, beralih ke direktori yang sebelumnya masih di antrian (yang urutannya lebih awal untuk diproses dibandingkan setiap direktori yang baru saja diakses), lalu dilakukan penelusuran sebagaimana sebelumnya. Begitu seterusnya hingga sebuah file yang dicari ditemukan, beberapa file ditemukan, atau antrian kosong.

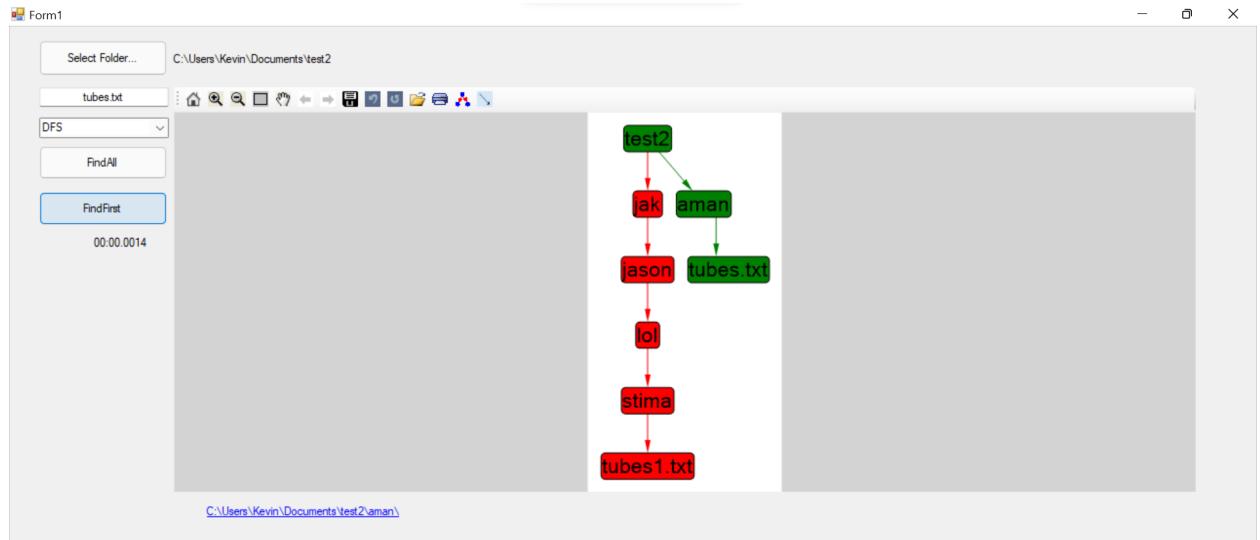
Sementara itu, pada algoritma DFS elemen utama yang digunakan adalah method SearchDFS. Terdapat kesamaan dalam penelusuran sebagaimana yang dieksekusi oleh SearchBFS, yakni prioritas terhadap setiap file untuk dikunjungi lebih dulu. Jika semua file telah dikunjungi, direktori pertama yang diproses akan langsung diakses setiap file dan direktorinya, lalu dilakukan pemrosesan terhadap file-file dan direktori dari direktori tersebut. Jika seluruh direktori sudah dikunjungi dan tidak memiliki sub direktori, akan dilakukan *backtrack* terhadap setiap direktori yang belum dikunjungi (yang ada pada antrian). Begitu seterusnya hingga sebuah file yang dicari ditemukan, beberapa file ditemukan, atau antrian kosong.

B. Menampilkan setiap file atau direktori sesuai spesifikasi tugas (warna jalur, status file, dll)

Ketika SearchBFS atau SearchDFS dijalankan, dalam memproses setiap file dan direktori, akan disimpan detil dari setiap folder atau file yang tertinjau, seperti siapa *parent*-ya, nama dirinya, status apakah dirinya sudah dikunjungi, belum dikunjungi, atau bukanlah target yang dicari, beserta urutan dirinya diakses. Dengan informasi-informasi tersebut, akan mudah divisualisasikan setiap file atau direktori berdasarkan spesifikasi tugas.

3.3 Contoh Ilustrasi Kasus Lain Yang Berbeda Dengan Contoh Pada Spesifikasi Tugas

Gambar berikut merupakan contoh ilustrasi kasus yang berbeda dengan spek tugas



Pada percobaan akan dicari file dengan nama tubes.txt dan dengan metode findfirst dan algoritma DFS. Maka alur program akan sebagai berikut test2 → jak → jason → lol → stima → aman → tubes1.txt → tubes.txt. Program akan mencari file pada prioritas hingga sangat dalam yaitu tubes1.txt dan kemudian melakukan backtracking simpul tetangga yang paling dekat, dalam kasus ini backtracking dilakukan pada simpul aman yang kemudian langsung ditemukan tubes.txt.

BAB 4

IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Program

Program dibagi menjadi parent class filesAndFolder yang memiliki dua turunan, DFS dan BFS. DFS memiliki atribut yang diperlukan untuk pencarian file menggunakan algoritma DFS, begitu juga BFS. kedua kelas memiliki method searchBFS/searchDFS untuk mencari file pada root directory yang diperlukan.

```
Public class filesAndFolder
{
    public string parent;
    public string direct;
    public string status;
    public int id;

    public filesAndFolder()
    {
        this.parent = "";
        this.direct = "";
        this.status = "queued";
        this.id = -999;
    }

    public filesAndFolder(string parent, string direct, string
status, int id)
    {
        this.parent = parent;
        this.direct = direct;
        this.status = status;
        this.id = id;
    }
}
```

```

public filesAndFolder(filesAndFolder fl)
{
    this.parent = fl.parent;
    this.direct = fl.direct;
    this.status = fl.status;
    this.id = fl.id;
}

public void setvalue(string parent, string direct)
{
    this.parent = parent;
    this.direct = direct;
}

class BFS : filesAndFolder
{
    private Graph graph;
    private Stopwatch stopwatch;
    public static string pathBFS = "";
    private List<string> listPathBFS;

    public Queue<filesAndFolder> nodeBFS = new
    Queue<filesAndFolder>(); // Queue buat output

    public BFS()
    {
        this.graph = new Graph();
        this.stopwatch = new Stopwatch();
        this.listPathBFS = new List<string>(10000);
    }

    public void pathFound(filesAndFolder found)

```

```

{
    int curr = found.id;
    while (curr != -1)
    {
        found.status = "found";
        string parentname = found.parent;
        found = getNodeByName(parentname);
        curr = found.id;
    }
}

public void pathFalse(filesAndFolder found)
{
    int curr = found.id;
    while (curr != -1 && found.status != "found")
    {
        found.status = "false";
        string parentname = found.parent;
        found = getNodeByName(parentname);
        curr = found.id;
    }
}
filesAndFolder getNodeByName(string name)
{
    foreach (filesAndFolder anak in nodeBFS)
    {
        if (anak.direct == name)
        {
            return (anak);
        }
    }
    return null;
}

```

```

    }

    public List<string> getListPathBFS()
    {
        return this.listPathBFS;
    }

    public string getTimeElapsed()
    {
        TimeSpan ts = this.stopwatch.Elapsed;

        return (ts.ToString(@"mm\:ss\.\ffff"));
    }

    public void SearchBFS(string root, string filename, bool
IsAllOccurrences)
{
    this.stopwatch.Start();
    Queue<string> dirs_visited = new Queue<string>(10000);

    if (!System.IO.Directory.Exists(root))
    {
        throw new ArgumentException();
    }

    dirs_visited.Enqueue(root);
    nodeBFS.Enqueue(new filesAndFolder("", root, "queued",
-1));
    int id = 1;
}

```

```

while (dirs_visited.Count > 0)
{
    string currentDir = dirs_visited.Dequeue();
    pathBFS = currentDir + "\\\";

    string[] files = null;
    try
    {
        files =
System.IO.Directory.GetFiles(currentDir);
    }

    catch (UnauthorizedAccessException e)
    {

        System.Console.WriteLine(e.Message);
        continue;
    }

    catch (System.IO.DirectoryNotFoundException e)
    {
        System.Console.WriteLine(e.Message);
        continue;
    }
    foreach (string file in files)
    {
        filesAndFolder proccess = new
filesAndFolder(currentDir, file, "queued", id);
        id++;
        nodeBFS.Enqueue(proccess);
    }
    foreach (string file in files)
}

```

```

{
    try
    {
        filesAndFolder proccess =
getNodeByName(file);

        System.IO.FileInfo fi = new
System.IO.FileInfo(file);

        if (fi.Name == filename)
        {
            pathFound(proccess);

            this.listPathBFS.Add(pathBFS);
            pathBFS += filename;
            System.Console.WriteLine(pathBFS);

            if (IsAllOccurences)
            {
                continue;
            }
            else
            {
                this.stopwatch.Stop();
                return;
            }
        }
        else
        {
            pathFalse(proccess);
        }
    }
    catch (System.IO.FileNotFoundException e)
    {
        System.Console.WriteLine(e.Message);
        continue;
    }
}

```

```

        }

    }

    string[] subDirs;
    try
    {
        subDirs = System.IO.Directory.GetDirectories(currentDir);
    }

    catch (UnauthorizedAccessException e)
    {
        System.Console.WriteLine(e.Message);
        continue;
    }

    catch (System.IO.DirectoryNotFoundException e)
    {
        System.Console.WriteLine(e.Message);
        continue;
    }

    foreach (string str in subDirs)
    {
        dirs_visited.Enqueue(str);
        nodeBFS.Enqueue(new filesAndFolder(currentDir,
str, "queued", id));
        id++;
    }
}

this.stopwatch.Stop();
}

public void createGraphBFS()

```

```

{
    foreach (filesAndFolder anak in nodeBFS)
    {
        foreach (filesAndFolder ortu in nodeBFS)
        {
            if (anak.parent == ortu.direct)
            {
                if (anak.status == "found")
                {
                    graph.AddEdge(ortu.direct,
anak.direct).Attr.Color =
Microsoft.Msagl.Drawing.Color.Green;

graph.FindNode(anak.direct).Attr.FillColor =
Microsoft.Msagl.Drawing.Color.Green;

graph.FindNode(anak.parent).Attr.FillColor =
Microsoft.Msagl.Drawing.Color.Green;
                    graph.FindNode(anak.parent).Label.Text =
new DirectoryInfo(anak.parent).Name;
                    graph.FindNode(anak.direct).Label.Text =
new DirectoryInfo(anak.direct).Name;
                    break;
                }
            else if (anak.status == "false")
            {
                graph.AddEdge(ortu.direct,
anak.direct).Attr.Color =
Microsoft.Msagl.Drawing.Color.Red;

graph.FindNode(anak.direct).Attr.FillColor =
Microsoft.Msagl.Drawing.Color.Red;

```

```

graph.FindNode(anak.parent).Label.Text =
new DirectoryInfo(anak.parent).Name;

graph.FindNode(anak.direct).Label.Text =
new DirectoryInfo(anak.direct).Name;

break;

}

else

{

    graph.AddEdge(ortu.direct, anak.direct);

    graph.FindNode(anak.parent).Label.Text =
new DirectoryInfo(anak.parent).Name;

    graph.FindNode(anak.direct).Label.Text =
new DirectoryInfo(anak.direct).Name;

break;

}

}

}

}

public Graph getGraphBFS()

{

    return this.graph;

}

}

public DFS()

{

    this.graph = new Graph();

    this.stopwatch = new Stopwatch();

    this.listPathDFS = new List<string>(10000);

}

```

```

public void pathFound(filesAndFolderDFS found)
{
    int curr = found.id;
    while (curr != -1)
    {
        found.status = "found";
        string parentname = found.parent;
        found = getNodeByNameDFS(parentname);
        curr = found.id;
    }
}

public void pathFalse(filesAndFolderDFS found)
{
    int curr = found.id;
    while (curr != -1 && found.status != "found")
    {
        found.status = "false";
        string parentname = found.parent;
        found = getNodeByNameDFS(parentname);
        curr = found.id;
    }
}

public List<string> getListPathDFS()
{
    return this.listPathDFS;
}

filesAndFolderDFS getNodeByNameDFS(string name)
{
    foreach (filesAndFolderDFS anak in nodeDFS)
    {

```

```

        if (anak.direct == name)
        {
            return (anak);
        }
    }

    return null;
}

public string getTimeElapsed()
{
    TimeSpan ts = this.stopwatch.Elapsed;

    return (ts.ToString(@"mm\:ss\.\ffff"));
}

public void SearchDFS(string root, string filename, bool
IsAllOccurrences)
{
    this.stopwatch.Start();
    Stack<string> dirs_visited = new Stack<string>(10000);

    if (!System.IO.Directory.Exists(root))
    {
        throw new ArgumentException();
    }

    int id = 1;
    dirs_visited.Push(root);
    nodeDFS.Enqueue(new filesAndFolderDFS("", root, "false",
-1));
}

while (dirs_visited.Count > 0)

```

```

{
    string currentDir = dirs_visited.Pop();
    pathDFS = currentDir + "\\\";

    string[] files = null;
    try
    {
        files = System.IO.Directory.GetFiles(currentDir);
    }

    catch (UnauthorizedAccessException e)
    {
        System.Console.WriteLine(e.Message);
        continue;
    }

    catch (System.IO.DirectoryNotFoundException e)
    {
        System.Console.WriteLine(e.Message);
        continue;
    }

    foreach (string file in files)
    {
        filesAndFolderDFS proccess = new
        filesAndFolderDFS(currentDir, file, "queued", id);
        nodeDFS.Enqueue(proccess);
        id++;
        try
        {

```

```

System.IO.FileInfo fi = new
System.IO.FileInfo(file);

if (fi.Name == filename)
{
    pathFound(proccess);
    this.listPathDFS.Add(pathDFS);
    System.Console.WriteLine(pathDFS);
    if (IsAllOccurences)
    {
        continue;
    }
    else
    {
        this.stopwatch.Stop();
        return;
    }
}
else
{
    pathFalse(proccess);
}
}

catch (System.IO.FileNotFoundException e)
{
    System.Console.WriteLine(e.Message);
    continue;
}
}

string[] subdirs;
try
{

```

```

                subdirs      =
System.IO.Directory.GetDirectories(currentDir);
}

catch (UnauthorizedAccessException e)
{
    System.Console.WriteLine(e.Message);
    continue;
}

catch (System.IO.DirectoryNotFoundException e)
{
    System.Console.WriteLine(e.Message);
    continue;
}

foreach (string str in subdirs)
{
    dirs_visited.Push(str);
    nodeDFS.Enqueue(new
filesAndFolderDFS(currentDir, str, "queued", id));
    id++;
}
this.stopwatch.Stop();
}

public void createGraphDFS()
{
    foreach (filesAndFolderDFS anak in nodeDFS)
    {
        foreach (filesAndFolderDFS ortu in nodeDFS)
        {
            if (anak.parent == ortu.direct)

```

```

    {

        if (anak.status == "found")
        {
            graph.AddEdge(ortu.direct,
anak.direct).Attr.Color =
Microsoft.Msagl.Drawing.Color.Green;

            graph.FindNode(anak.direct).Attr.FillColor =
Microsoft.Msagl.Drawing.Color.Green;

            graph.FindNode(anak.parent).Attr.FillColor =
Microsoft.Msagl.Drawing.Color.Green;
            graph.FindNode(anak.parent).Label.Text =
new DirectoryInfo(anak.parent).Name;
            graph.FindNode(anak.direct).Label.Text =
new DirectoryInfo(anak.direct).Name;
            break;
        }
        else if (anak.status == "false")
        {
            graph.AddEdge(ortu.direct,
anak.direct).Attr.Color =
Microsoft.Msagl.Drawing.Color.Red;

            graph.FindNode(anak.direct).Attr.FillColor =
Microsoft.Msagl.Drawing.Color.Red;
            graph.FindNode(anak.parent).Label.Text =
new DirectoryInfo(anak.parent).Name;
            graph.FindNode(anak.direct).Label.Text =
new DirectoryInfo(anak.direct).Name;
            break;
        }
    }

```

```

        }
    }

}

public Graph getGraphDFS()
{
    return this.graph;
}

}

```

4.2 Spesifikasi Program dan Struktur Data

Pada Program ini, kami menggunakan kelas filesandfolder berupa Queue of Node pada pengimplementasian pencarian BFS serta DFS. Queue of Node ini memiliki atribut sebagai berikut:

1. parent(string): berupa nama induk dari node terkait
2. direct(string): berupa nama node
3. status(string): berupa status pencarian (“queued” berarti masih dalam antrian, “found ” berarti merupakan path yang benar, “fakse”, berarti path salah)
4. id(int): pembeda antara masing-masing node

Lalu pada kelas filesandfolder ini memiliki kelas turunan, yaitu BFS dan DFS. Perbedaan dari BFS dan DFS adalah BFS menggunakan struktur data Queue dalam proses pencarian sehingga menganut sistem first in first out (files yang masuk antrian akan diperiksa sesuai urutan) sedangkan DFS menggunakan stack sehingga menganut first in last out. File yang masuk ke dalam stack terlebih dahulu akan diproses paling terakhir. Kelas DFS dan BFS memiliki atribut sebagai berikut:

1. graph(Graph): graf dari DFS
2. stopwatch(Stopwatch): waktu pencarian algoritma
3. pathBFS/pathDFS(string): path pencarian sementara
4. ListPath(list of string): list berupa path ditemukannya file

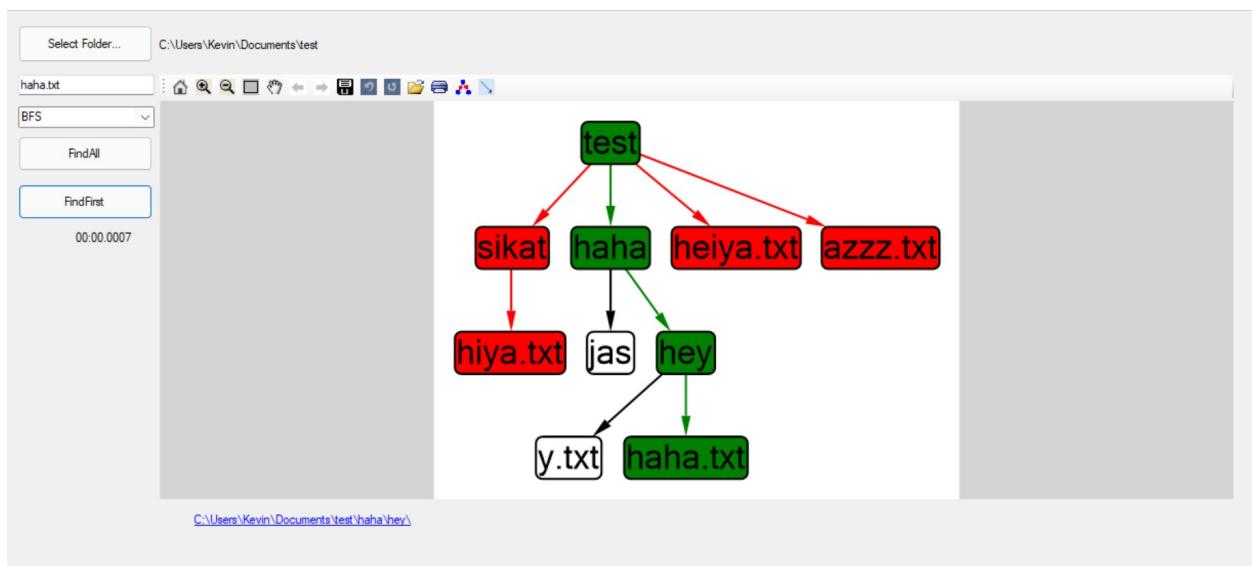
Program dibuat menggunakan kakas Visual Studio .NET dan menggunakan C# Desktop Development. Pembuatan graf menggunakan kakas MSAGL yang dapat digunakan pada Visual Studio. Program hanya dapat digunakan untuk mencari nama file.

4.3 Penjelasan dan Tata Cara Penggunaan Program

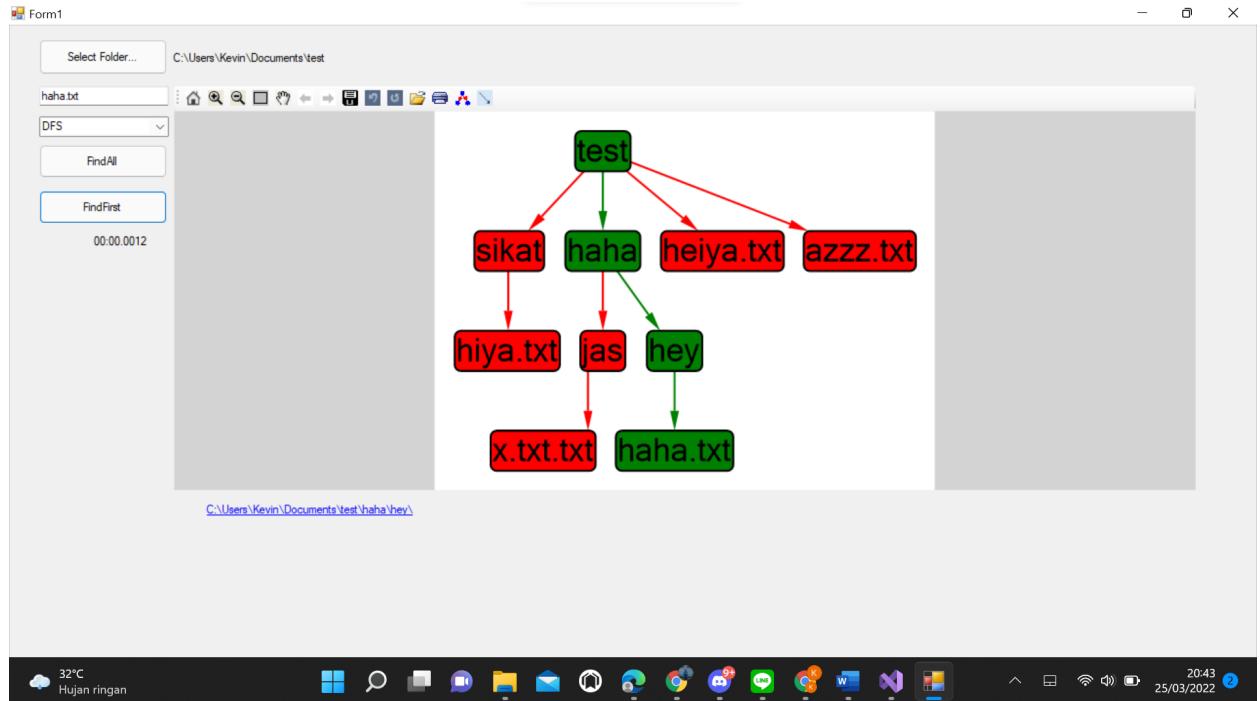
Program yang kami buat memiliki fitur-fitur sebagai berikut:

1. Select Folder untuk memilih root directory
2. Input filename yang akan dicari
3. Pilih metode pencarian (BFS atau DFS)
4. Pilih opsi pencarian (mencari semua atau cukup 1 kali kemunculan)
5. Graf akan muncul di sebelah kanan panel input, waktu eksekusi akan muncul di bawah tombol “FindFirst”, gambar akan muncul di sebelah kanan panel input, dan hyperlink dapat ditekan di bawah gambar grafik.

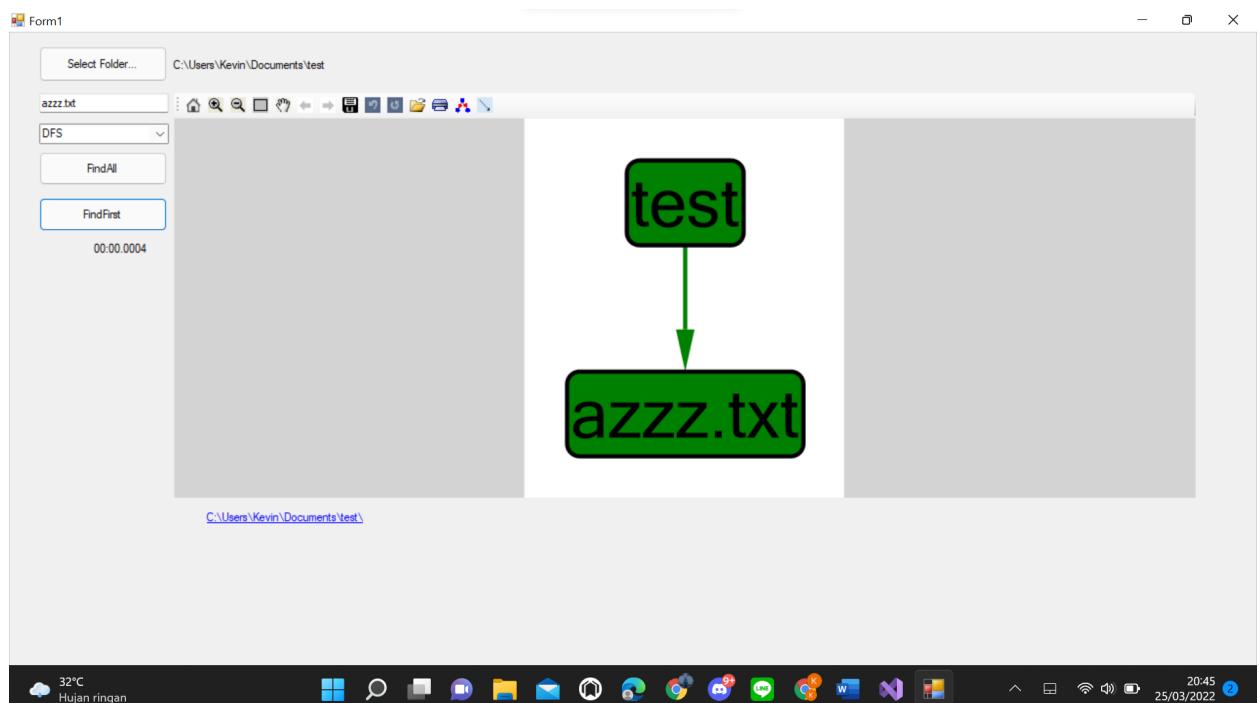
4.4 Hasil Pengujian



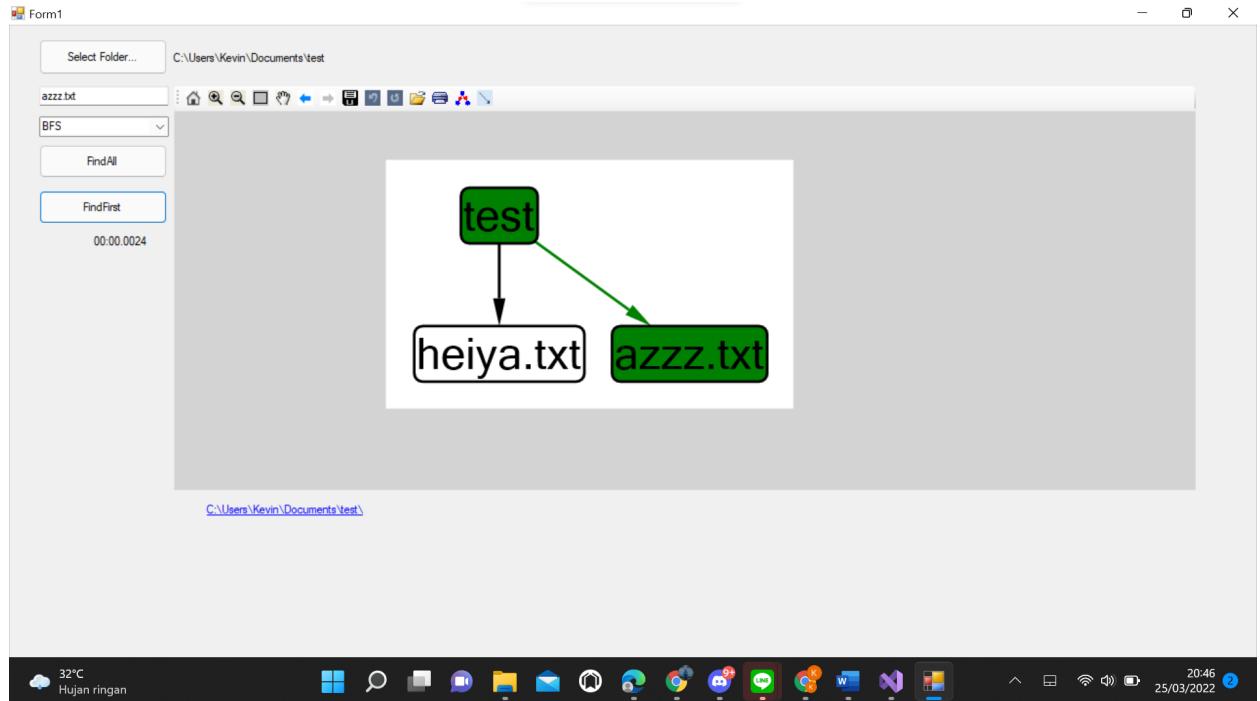
Pencarian haha.txt dengan BFS dan hanya dicari untuk 1 file pertama



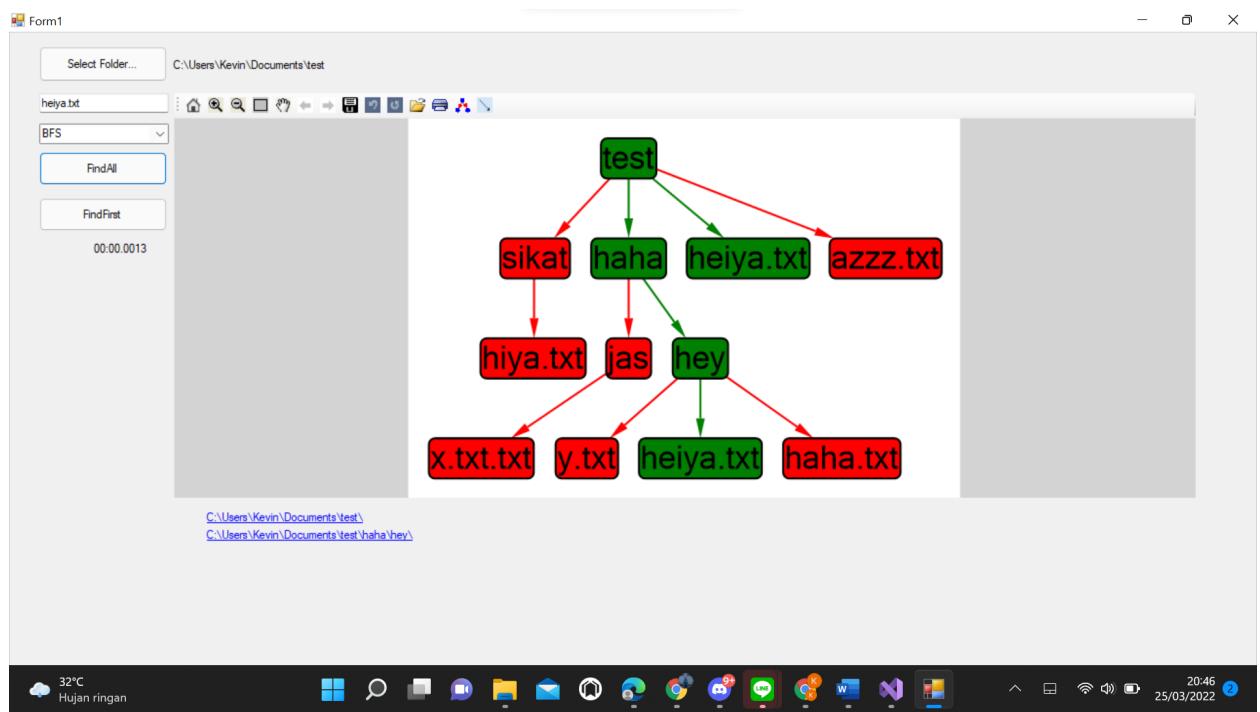
Pencarian haha.txt pada root folder hey dengan menggunakan dfs dan hanya dicari 1



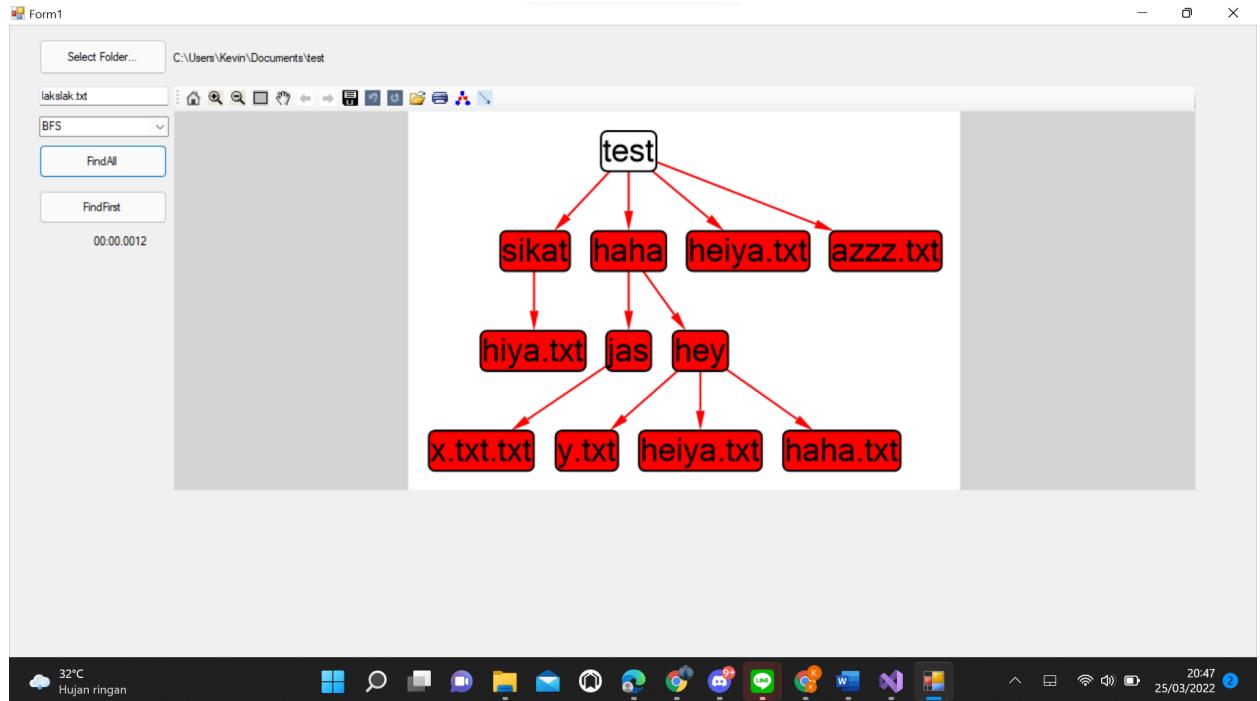
Pencarian azzz.txt dengan metode DFS dan hanya mencari kemunculan pertama



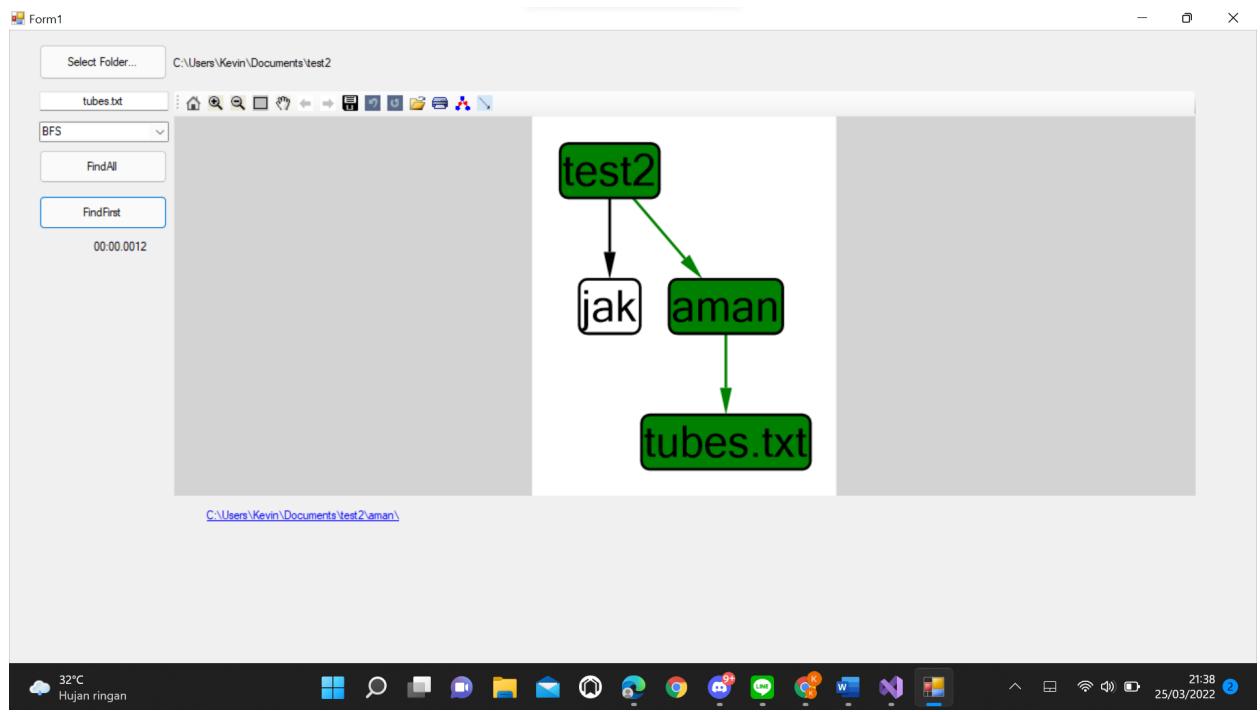
Pencarian azzz.txt dengan menggunakan BFS dan hanya 1 kemunculan

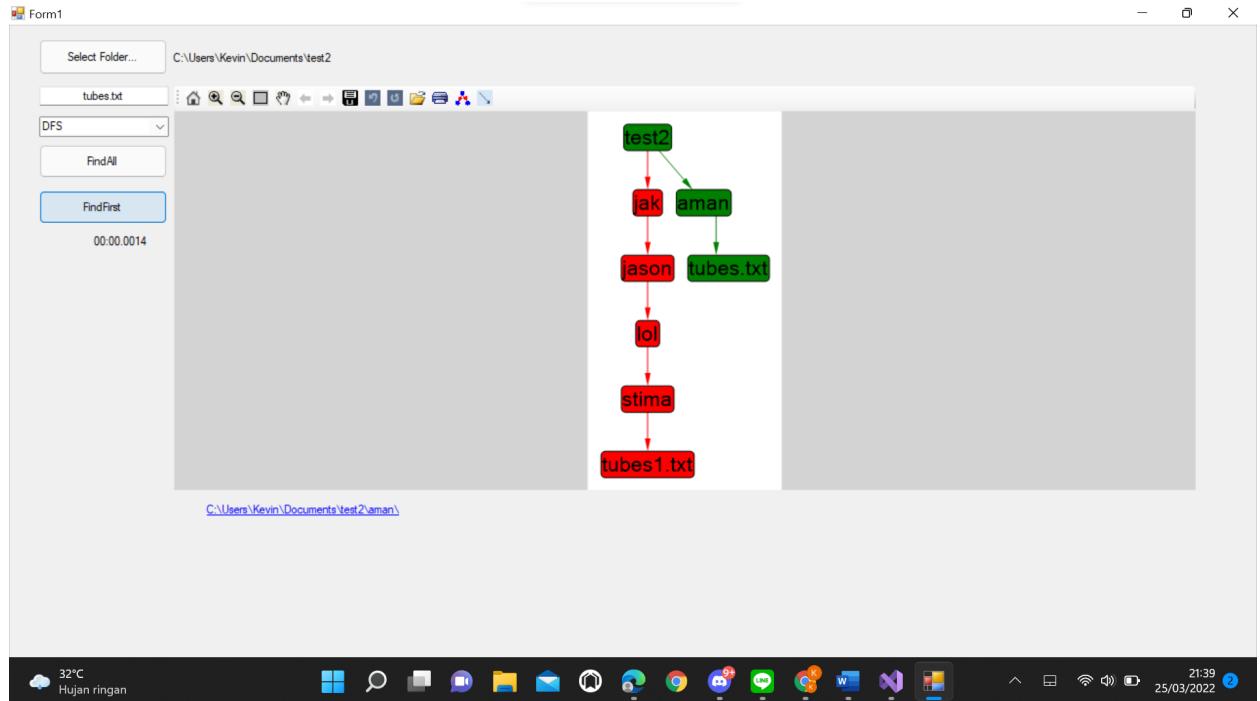


Contoh program menghasilkan dua kemunculan dengan FindAll

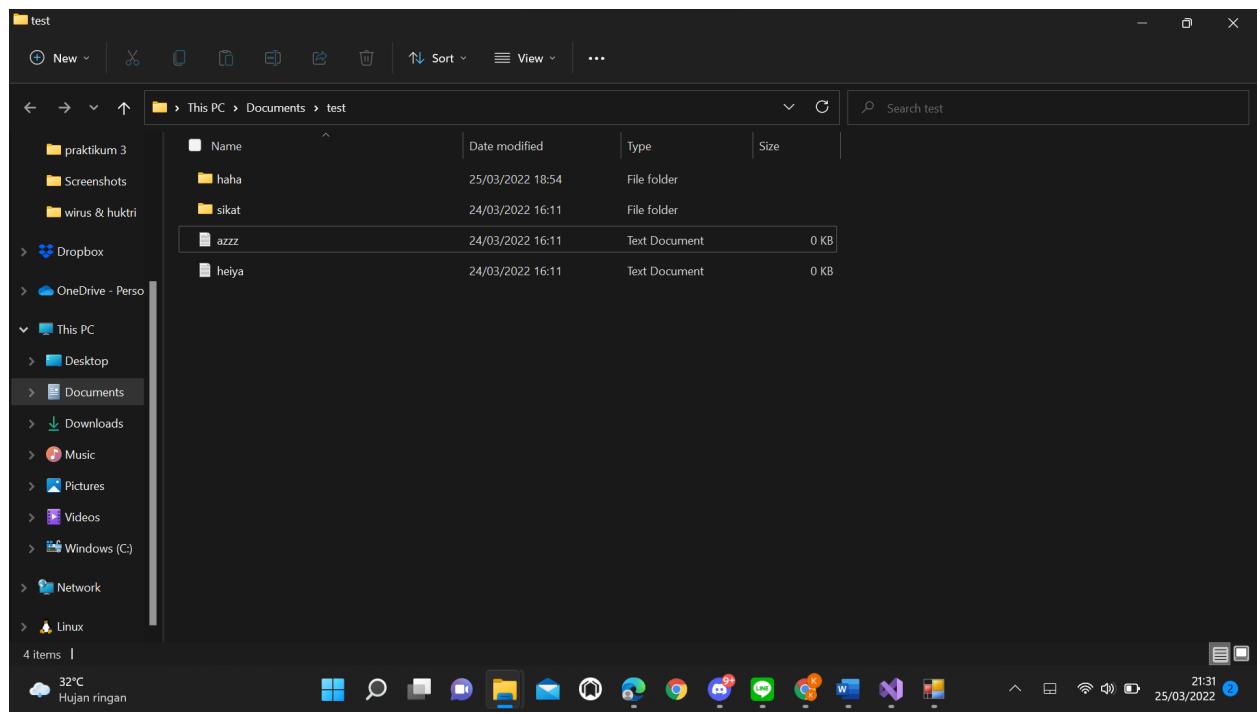


Contoh output program jika tidak menemukan file yang dicari





Perbedaan mencari tubes.txt untuk dua metode berbeda



Tampilan ketika hyperlink ditekan

4.5 Analisis Desain Solusi Algoritma BFS dan DFS

Berdasarkan hasil percobaan, jika dilihat ketika mencari file yang berada dekat dengan root dir

ector, pencarian DFS memiliki kemungkinan dapat yang sangat fluktuatif, bisa sangat cepat, bisa sangat lambat. Pada pencarian tubes.txt dapat dilihat bahwa ketika algoritma DFS dijalankan, program akan mencari sangat jauh ke dalam sebelum melakukan backtracking ke directory yang sebenarnya file tubes.txt ada. Sedangkan pada pencarian BFS, karena tubes.txt berada pada kedalaman yang tidak begitu dalam, algoritma BFS selesai mencari dalam 0:0012 detik, sedangkan algoritma DFS selesai dalam 0:0014 detik. Bukan perbedaan waktu yang besar, tetapi jika ukuran semakin diperbesar maka perbedaan waktu akan semakin terlihat.

Akan tetapi, DFS juga mungkin untuk selesai dengan sangat cepat. Pada pencarian azzz.txt, algoritma DFS langsung mengecek prioritas file, yaitu yang sesuai dengan abjad dan langsung menemukan file azzz.txt. Sedangkan pada algoritma BFS, algoritma akan membuat antrian terlebih dahulu (tidak langsung mengecek azzz.txt) sehingga pencarian memakan waktu lebih lama. Perbandingan waktu pencarian DFS bisa sangat cepat (hingga 6x lebih cepat pada kasus mencari azzz.txt) dibandingkan BFS.

Jika dianalisis secara mendalam, algoritma BFS memiliki efisiensi yang relatif lebih stabil, tergantung kedalaman file yang ingin dicari. Dapat dikatakan lama waktu pencarian menggunakan algoritma BFS akan sangat bergantung pada kedalaman file yang akan dicari, jika file sangat dalam maka tidak mungkin algoritma BFS akan mendapatkannya dengan cepat. Akan tetapi, jika menggunakan algoritma DFS, lama waktunya akan bergantung terhadap prioritas proses yang dimiliki algoritma. Seperti pada algoritma yang kami buat, karena menggunakan stack pada proses pemilihan directorynya, semakin path berada pada folder yang memiliki abjad awal (a: awal, Z: akhir), maka pencarian akan semakin lama. Sedangkan pada file yang berada pada folder abjad akhir, pencarian akan bisa sangat cepat.

BAB 5

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Pada Tugas Besar kali ini, kami memperoleh pengetahuan bahwa algoritma pencarian solusi dengan metode BFS dan DFS dapat dengan efektif digunakan untuk pembuatan program *Folder Crawler* dengan memanfaatkan bahasa pemrograman C# untuk pembuatan aplikasi *desktop*-nya. Program ini menggambarkan pencarian dengan visualisasi berupa *tree* dengan simpul berupa *folder* dan *file* yang telah atau akan diperiksa, kami memberi pembeda yaitu pada warna simpulnya. Warna merah menandakan simpul telah diperiksa, warna hijau menandakan simpul telah diperiksa dan merupakan rute yang tepat menuju file yang dituju, dan tidak diwarnai jika simpul belum diperiksa. Algoritma pencarian BFS dan DFS berbeda pada bagian struktur datanya, BFS menggunakan struktur data *queue* sedangkan DFS menggunakan struktur data *stack*.

5.2 Saran

Saran dari kami untuk kedepannya adalah sebaiknya requirement penggerjaan Tugas Besar tidak terestriksi hanya untuk satu OS (*Operating System*) saja karena dapat menimbulkan kesulitan komunikasi atau kerja sama antar anggota kelompok jika salah satu anggotanya menggunakan device dengan OS yang tidak compatible dengan yang diminta di spesifikasi.

DAFTAR PUSTAKA

Munir, Rinaldi. (2022). Spesifikasi Tugas Besar.

https://cdn-edunex.itb.ac.id/38015-Algorithm-Strategies-Parallel-Class/85259-BFS-dan-DFS/1646201812962_Tugas-Besar-2-IF2211-Strategi-Algoritma-2022.pdf (diakses tanggal 18 Maret 2022).

(Spesifikasi Tugas Besar 2)

Munir, Rinaldi. (2022). Materi Metode BFS dan DFS.

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf> (diakses tanggal 18 Maret 2022).

(Materi tentang metode BFS dan DFS.

LAMPIRAN

Repository Github

https://github.com/jakartasipirok/Tubes2_13519210

Video Demonstrasi

<https://youtu.be/lLaBvJiw67o>