

Tugas Besar 2 IF2211 Strategi Algoritma

Pengaplikasian Algoritma BFS dan DFS dalam Implementasi

Folder Crawling



Disusun oleh:

cool story

13520016 - Gagas Praharsa Bahar

13520044 - Adiyansa Prasetya Wicaksana

13520081 - Andhika Arta Aryanto

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

2022

DAFTAR ISI

BAB 1	3
DESKRIPSI MASALAH	3
BAB 2	4
TEORI SINGKAT	4
Traversal Graph	4
Tree	4
Algoritma Breadth First Search (BFS)	5
Algoritma Depth First Search (DFS)	5
C# Desktop Application Development	6
BAB 3	8
ANALISIS PEMECAHAN MASALAH	8
Langkah-langkah Pemecahan Masalah	8
Mapping Persoalan dalam BFS dan DFS	8
Contoh Ilustrasi Kasus	10
BAB 4	14
IMPLEMENTASI DAN PENGUJIAN	14
Repository Github dan Video Penjelasan	14
Implementasi Dalam Pseudocode	14
Struktur Data Program	16
Tata Cara Penggunaan Program	18
Pengujian dan Hasil	21
Analisis	37
BAB 5	39
KESIMPULAN DAN SARAN	39
Kesimpulan	39
Saran	39
DAFTAR PUSTAKA	40

BAB 1

DESKRIPSI MASALAH

Folder crawling adalah sebuah aksi di mana mesin komputer akan mulai mencari file yang sesuai dengan query mulai dari starting directory hingga seluruh children dari starting directory tersebut sampai satu file pertama/seluruh file ditemukan atau tidak ada file yang ditemukan. Algoritma yang dapat dipilih untuk melakukan crawling tersebut pun dapat bermacam-macam dan setiap algoritma akan memiliki teknik dan konsekuensinya sendiri. Oleh karena itu, penting agar komputer memilih algoritma yang tepat sehingga hasil yang diinginkan dapat ditemukan dalam waktu yang singkat.

Dalam tugas besar ini, kami diminta untuk membangun sebuah aplikasi GUI sederhana dengan bahasa C# 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 diinginkan dan memvisualisasikan hasil dari pencarian folder tersebut dalam bentuk pohon.

Selain pohon, kami 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.

BAB 2

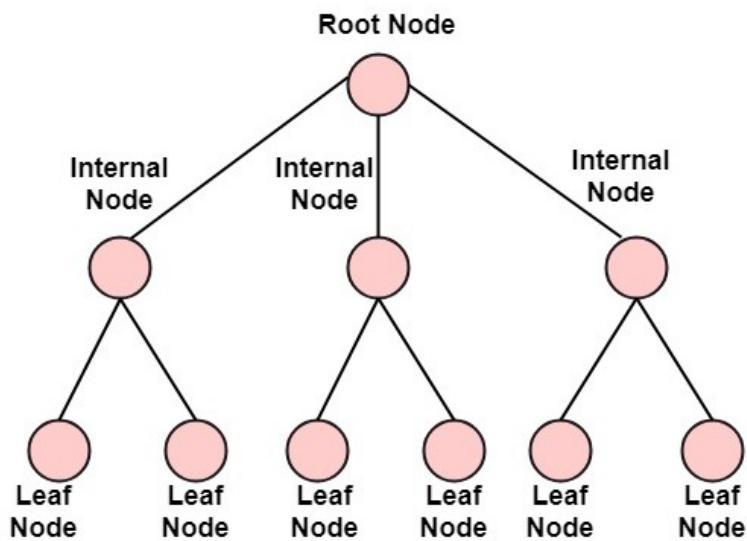
TEORI SINGKAT

2.1 Traversal Graph

Traversal graf atau *graph traversal* adalah proses mengunjungi setiap simpul pada sebuah graf dengan cara yang sistematis. Algoritma-algoritma traversal graf diklasifikasikan atas bagaimana cara algoritma tersebut menentukan urutan kunjungan untuk setiap simpul pada graf. Beberapa dari algoritma traversal graf dapat diklasifikasikan menjadi dua buah algoritma berbasis pohon, yaitu *Breadth First Search* (BFS) yang berarti pencarian melebar dan *Depth First Search* yang berarti pencarian mendalam.

2.2 Tree

Tree (pohon) merupakan graf tak-berarah terhubung yang tidak mengandung sirkuit. *Rooted Tree* (pohon berakar) merupakan pohon yang satu buah simpulnya diperlakukan sebagai akar dan sisi-sisinya diberi arah sehingga menjadi sebuah graf berarah. Untuk mempersingkat penamaan, istilah *tree* (pohon) yang digunakan selanjutnya akan merujuk kepada *rooted tree* (pohon berakar).

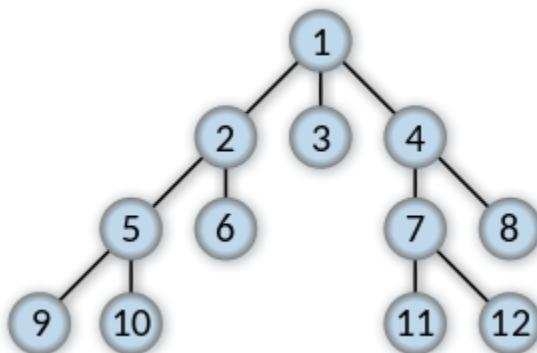


Gambar 2.2.1 Komponen dalam Pohon Berakar

Komponen yang dimiliki pada *tree* sesuai dengan gambar 2.2.1 adalah *root node* yang merupakan simpul yang diperlukan sebagai akar, *internal node* merupakan sisi yang memiliki sisi (*internal node*) lain sebagai anak, dan yang terakhir *leaf node* merupakan sisi yang tidak memiliki anak (tetangga).

2.3 Algoritma *Breadth First Search* (BFS)

Algoritma *Breadth First Search* atau yang biasa disingkat BFS adalah algoritma traversal graf yang memiliki ciri utama pencarian yang melebar. BFS dan aplikasinya pada graf ditemukan oleh Konrad Zuse pada 1945 pada tesisnya yang tidak dikemukakan, dan ditemukan kembali oleh Edward F. Moore pada 1959 yang mengimplementasikannya untuk mencari jalan tercepat keluar dari sebuah labirin.

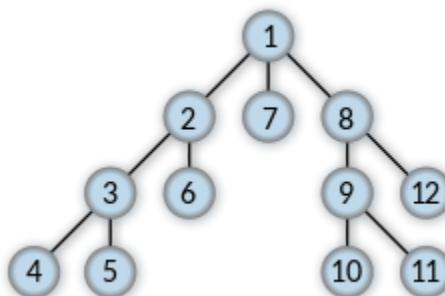


Pada implementasi BFS, pencarian dimulai dari simpul akar, lalu mengunjungi semua simpul yang berada pada sebuah kedalaman tertentu sebelum masuk ke simpul-simpul yang berada pada kedalaman berikutnya. Implementasi BFS seringkali disertakan dengan implementasi *queue*, yang digunakan sebagai penampung simpul anak yang sudah terlihat namun belum dikunjungi.

2.4 Algoritma *Depth First Search* (DFS)

Algoritma *Depth First Search* atau yang biasa disingkat DFS adalah algoritma traversal graf yang memiliki ciri utama pencarian yang mendalam, memprioritaskan simpul yang berada

pada kedalaman lebih tinggi. DFS ditemukan oleh matematikawan asal Perancis, Charles Pierre Trémaux pada abad ke 19 sebagai strategi untuk menyelesaikan labirin.

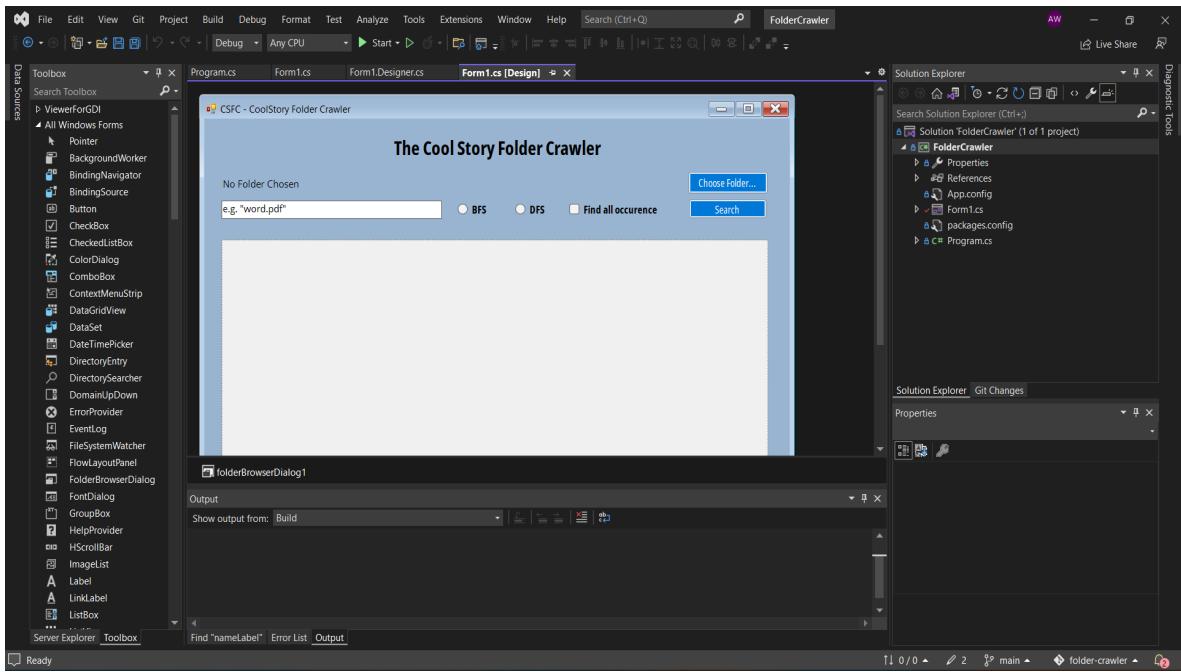


Pada implementasi DFS, pencarian dimulai dari simpul akar, lalu mengunjungi simpul pada kedalaman selanjutnya dan dari simpul ini dilanjutkan ke simpul yang lebih dalam lagi sampai mencapai yang terdalam. Implementasi dari DFS ini seringkali digunakan dalam bentuk rekursif karena bentuk dari DFS yang memanggil simpul berikutnya yang lebih dalam secara terus menerus ataupun bisa diimplementasikan menggunakan *stack*.

2.5 C# Desktop Application Development

C# (dibaca “See Sharp”) merupakan bahasa pemrograman yang modern, *object-oriented*, dan *type-safe*. C# menggunakan .NET Framework yang merupakan *framework* pengembangan perangkat lunak pada suatu runtime yang paling predominan digunakan dalam platform Windows. Framework ini memiliki akses ke library kelas-kelas yang sangat ekstensif (FCL) dan juga menawarkan *common language runtime* yang menyediakan *virtual machine* sebagai lapisan perlindungan keamanan lebih dan berperan untuk *exception handling* program.

Bahasa C# dikembangkan oleh Microsoft pada tahun 2000 serta dirancang oleh Anders Hejlsberg. C# dibuat dengan dasar dari keluarga bahasa pemrograman C sehingga *syntax* yang dimilikinya akan mirip dengan bahasa C, C++, Java, dan JavaScript. Bahasa C# karena merupakan bahasa yang *object-oriented* menerapkan konsep dasar pemrograman *object-oriented* seperti *inheritance*, enkapsulasi, dan polimorfisme. C# mengutamakan *versioning* untuk memastikan program dan *library* dapat berkembang secara terus menerus, dengan salah satu aspek yang digunakannya adalah penggunaan *override* dan *kelas virtual*.



Gambar 2.4.1 Contoh Penggunaan Bahasa C# dalam Desktop Application Development

Bahasa C# biasanya digunakan dengan IDE bawaannya yaitu Visual Studio untuk mempermudah penggunaan. C# dengan Visual Studio seperti pada Gambar 2.4.1 mampu membuat GUI *Desktop Application* dengan cukup simple. Peletakan komponen dalam GUI dapat dilakukan dengan cara *drag and drop* dari Toolbox, sehingga memudahkan dalam melakukan *development* dibanding yang biasanya dilakukan dengan *hard-code*.

BAB 3

ANALISIS PEMECAHAN MASALAH

3.1. Langkah-langkah Pemecahan Masalah

Permasalahan yang akan diselesaikan dalam Tugas Besar 2 ini adalah pencarian file dalam struktur *folder*. Pencarian dilakukan dengan memilih *folder* awal tempat pencarian dilakukan dan nama *file* yang akan dicari. Dalam pemecahan masalahnya akan digunakan algoritma pencarian DFS serta BFS.

Representasi dari *folder* awal pencarian dilakukan dan *file* yang akan dicari dapat direpresentasikan sebagai *tree*. Tiap komponen dari *tree* akan disebut sebagai *node* dan penghubung antara satu *node* ke *node* yang lain disebut sebagai *edges*. Struktur dari folder awal pencarian (*root*) memiliki satu akar (*root node*) dan banyak anak (*subdirectory* dan *file*). Dalam pemecahan masalahnya dibedakan menjadi dua cara yaitu BFS dan DFS. Implementasi BFS menggunakan metode iteratif dan struktur data *queue* sedangkan DFS menggunakan metode rekursif dan struktur data *array*.

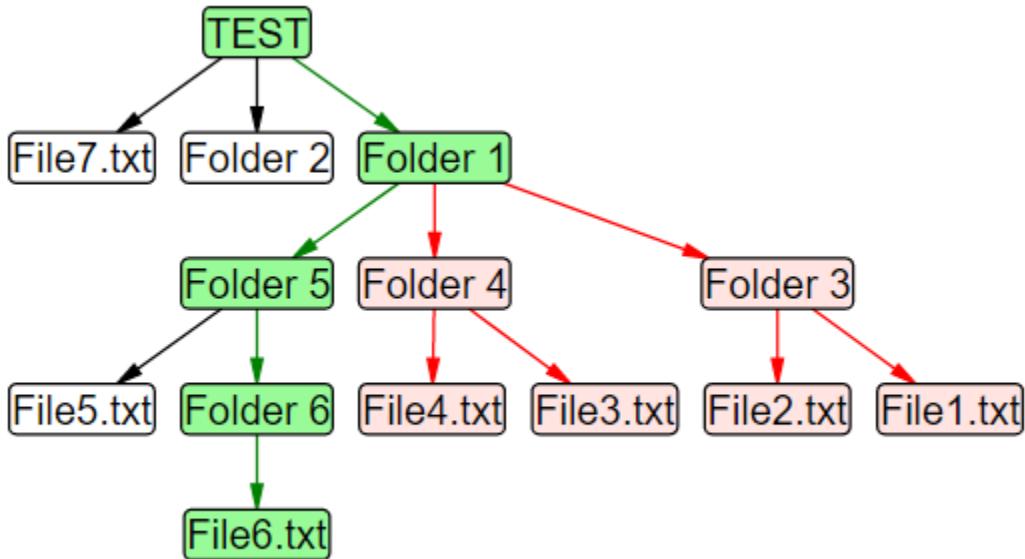
Permasalahan lainnya pada tugas besar ini adalah penggunaan Graphical User Interface (GUI) dalam program ini, visualisasi proses pencarian dalam bentuk *tree*, dan juga menampilkan *hyperlink* untuk kepada *folder* tempat *file* ditemukan. Permasalahan perlu diselesaikan dengan menggunakan bahasa pemrograman C# yang merupakan sebuah kakas untuk mengembangkan *desktop application* (aplikasi desktop) yang dilengkapi dengan .NET *framework*. Dengan menggunakan berbagai macam library yang disediakan dalam library bawaan C# dan juga .NET *framework*, proses pencarian pencarian dalam bentuk *tree* dapat menggunakan MSAGL dan pembuatan GUI dan *hyperlink* menggunakan Visual Studio.

3.2. Mapping Persoalan dalam BFS dan DFS

Berdasarkan analisis pemecahan masalah yang telah dilakukan, *mapping* persoalan ke dalam algoritma secara umum adalah sebagai berikut:

- a. *Root Folder* : *Root Node* dari *tree*
- b. *Subdirectory* : *Internal Node* dari *tree*

c. *File* : LeafNode dari tree



Gambar 3.2.1 Representasi Struktur Folder sebagai Tree

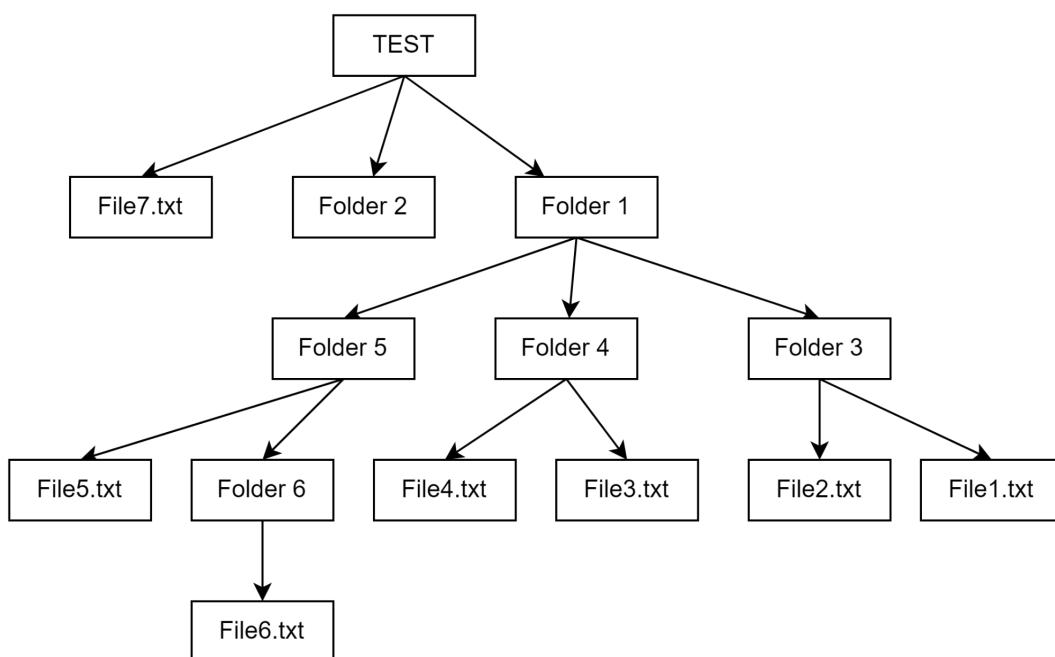
Setelah melakukan *mapping* persoalan tersebut, berikut merupakan rancangan singkat dari algoritma BFS dan DFS yang akan diimplementasikan.

- Algoritma BFS menggunakan struktur data *queue* sebagai struktur data yang lebih “simple” dari Tree. Metode yang dilakukan terhadap struktur datanya adalah iteratif. *Root Folder* akan di-*mapping* sebagai elemen pertama dari *queue*, *subdirectory* sebagai elemen dari *queue*, dan *file* sebagai *array*. Dalam pencarian *file*, untuk setiap *subdirectory* terhadap *root folder* akan dimasukkan ke dalam *queue* dan setiap *file* yang terdapat di *root folder* ataupun *subdirectory* akan dilakukan iterasi untuk mencocokkan dengan *file* target. Dan seterusnya, *subdirectory* akan menjadi root yang baru hingga *file* target ditemukan ataupun sampai semua *file* telah dikunjungi.
- Algoritma DFS menggunakan struktur data *array* yang juga merupakan struktur data yang lebih “simple” dibanding Tree. Metode yang digunakan adalah rekursif. Fungsi DFS menggunakan parameter *root folder* dan juga *file* target. Pada pemanggilannya, akan dibuat *array* yang berisi *subdirectory* dari *root folder* dan *array* yang berisi *file* dari *root folder*. Untuk setiap *subdirectory*, akan dilakukan pemanggilan DFS dengan *subdirectory*

menjadi parameter *root*-nya dan *file* target tetap. Dan untuk setiap *file*, akan dilakukan pencocokan dengan *file* target-nya. Pemanggilan akan terus dilakukan jika *file* sudah ditemukan atau pun jika semua *folder* dan *file* telah dikunjungi semua.

3.3. Contoh Ilustrasi Kasus

Untuk memperjelas analisis pemecahan masalah akan dilakukan ilustrasi kasus dengan penyelesaiannya sesuai dengan algoritma yang telah dirancang. Akan dilakukan pencarian terhadap *file* “File6.txt” terhadap *root folder* “TEST” dengan struktur *folder*-nya sebagai berikut.



Gambar 3.3.1 Stuktur Folder Ilustrasi Kasus

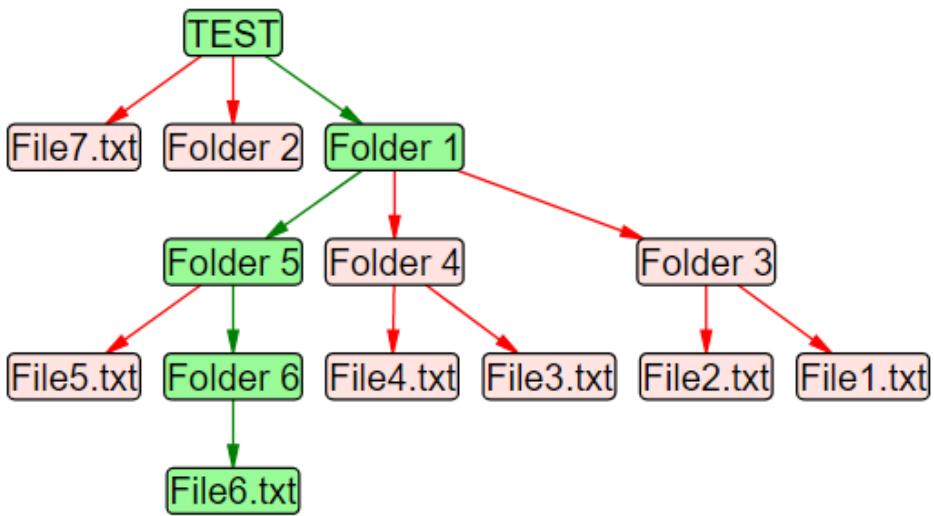
Pertama, akan dilakukan dengan menggunakan metode BFS. Elemen pertama yang masuk ke *queue* adalah “TEST” dengan *subdirectory*-nya yaitu “Folder 1” dan “Folder 2” lalu dilakukan pengecekan apakah pada “TEST” terdapat “File6.txt” atau tidak, karena pada kasus ini tidak ada, pencarian akan terus dilanjutkan. Elemen head/pertama dari queue akan diambil, yaitu “Folder 1” dan seperti sebelumnya semua *subdirectory* di dalamnya dimasukkan ke dalam queue,

karena di sini “File6.txt” tidak ada , pencarian terus dilakukan. Sekarang isi queue adalah [Folder 2, Folder 3, Folder 4, Folder 5]. Dequeue terus dilakukan, karena pada Folder 2, Folder 3, dan Folder 4 tidak terdapat *subdirectory* maupun “File6.txt” , tidak ada Folder baru yang dimasukkan ke dalam queue dan pencarian terus dilakukan. Sekarang, pada Folder 5 akan dilakukan queue untuk *subdirectory* Folder 6 dan saat pengecekan file juga tidak ketemu. Sekarang isi queue adalah [Folder 6], dilakukan deque lalu pengecekan file dan pada akhirnya disini “File6.txt” ditemukan.

TABEL BFS

HEAD	ISI QUEUE SEKARANG	FILE YANG DICEK
TEST	[Folder 1, Folder 2]	File7.txt
Folder 1	[Folder 2, Folder 3, Folder 4, Folder 5]	-
Folder 2	[Folder 3, Folder 4, Folder 5]	-
Folder 3	[Folder 4, Folder 5]	File1.txt, File2.txt
Folder 4	[Folder 5]	File3.txt, File4.txt
Folder 5	[Folder 6]	-
Folder 6	-	File6.txt (KETEMU!)

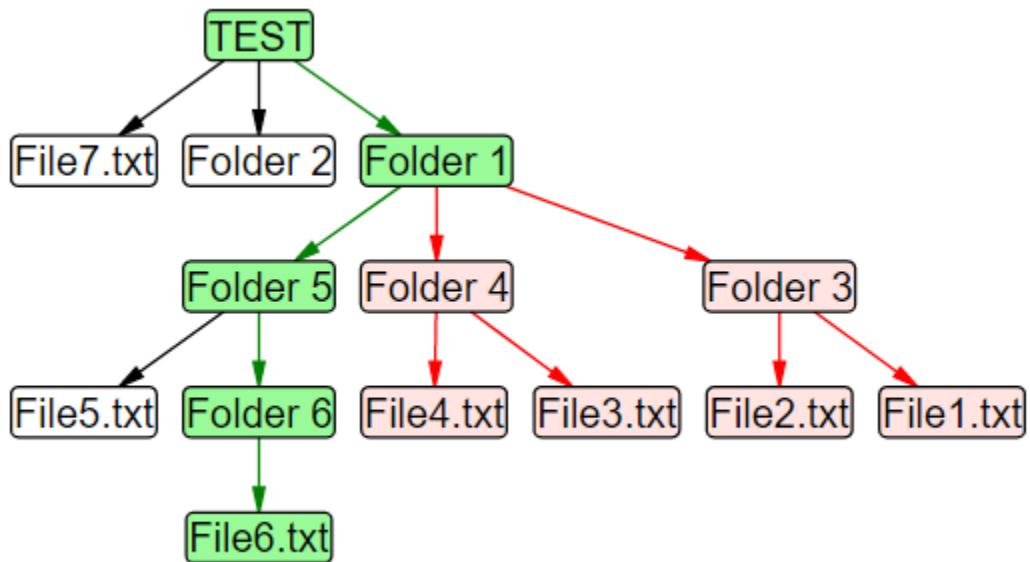
Tabel 3.3.1 Ilustrasi Tabel untuk BFS



Gambar 3.3.2 Tree dengan Algoritma BFS

Untuk cara kedua, akan dilakukan dengan metode DFS. *Root folder* yang menjadi parameter adalah “TEST” dan *file* targetnya adalah “File6.txt”. *Subdirectory* dari “TEST” adalah “Folder 1” dan “Folder 2” sedangkan *files* yang terdapat di “TEST” adalah File7.txt”. Algoritma DFS yang digunakan adalah dengan metode rekursif sehingga untuk “Folder 1” akan dilakukan pemanggilan fungsi DFS dengan “Folder 1” sebagai argumen fungsi. “Folder 1” memiliki *subdirectory* “Folder 3”, “Folder 4”, dan “Folder 5”, tetapi tidak memiliki *files*. Untuk “Folder 3” akan dilakukan pemanggilan DFS lagi dengan “Folder 3” sebagai argumen dari fungsinya.

“Folder 3” tidak mempunyai *subdirectory* sehingga tidak dilakukan pemanggilan fungsi DFS lagi untuk *subdirectory*-nya, tetapi mempunyai dua *file* yaitu “File1.txt” dan “File2.txt”, karena tidak dilakukan pemanggilan lagi pada rekursi ini maka akan dilakukan pengecekan terhadap “File1.txt” dan “File2.txt”. *File* target belum ditemukan sehingga akan dilakukan rekursi berikutnya yaitu terhadap “Folder 4” karena “Folder 3” tidak melakukan rekursi lagi dan sudah dilakukan pengecekan terhadap *leaf nodes*-nya. Rekursi akan terus dilakukan sampai rekursi menggunakan “Folder 6” sebagai *root*-nya dan menemukan “File6.txt” di *subdirectory* tersebut.



Gambar 3.3.3 Tree dengan Algoritma DFS

Rute pencarian yang dilakukan berdasarkan gambar 3.3.3 adalah sebagai berikut “TEST” → “Folder 1” → “Folder 3” → “File1.txt” → “File2.txt” → “Folder 4” → “File3.txt” → “File4.txt” → “Folder5” → “Folder6” → “File6.txt”. Warna merah pada gambar 3.3.3 merupakan rute pencarian yang telah dilakukan tetapi tidak ditemukan *file* targetnya, sedangkan warna hijau berarti rute pencarian yang menemukan *file* targetnya, sedangkan warna hitam merupakan rute pencarian yang selanjutnya akan dilakukan tetapi *file* target sudah lebih dahulu ditemukan.

BAB 4

IMPLEMENTASI DAN PENGUJIAN

4.1 Repository Github dan Video Penjelasan

Program yang kelompok kami terdapat pada repository Github dengan tautan [berikut](#) dan link video penjelasan terdapat pada tautan https://youtu.be/6_t7lseQ6eg.

4.2 Implementasi Dalam *Pseudocode*

A. Pseudocode Algoritma BFS

Pada bagian ini, akan dicantumkan pseudocode algoritma BFS yang dilakukan. Karena pada kode kami pembangkitan *graph* dilakukan di dalam Algoritma tersebut, pseudocode juga akan memasukkan pembangkitan graph

```
procedure BFS (input root : string, filename : string, SearchAll : bool)
{
    Melakukan traversal pada directory root menggunakan algoritma
    pencarian BFS

    Masukan : root adalah directory awal mulai pencarian, filename
    adalah nama file yang dicari, SearchAll adalah penanda apakah pengguna
    ingin mencari semua kemunculan file
    dalam directory tersebut

    Keluaran : graph hasil traversal dimunculkan ke layar dengan rute ke
    file yang dicari
    Berwarna hijau, rute yang dikunjungi namun tidak memiliki file
    merah, dan rute yang
    Sudah berada di queue namun tidak sempat dicek diwarnai hitam
}
Deklarasi
q : antrian directory
daftarfile, daftarsubdir : array berisi string
queuehead : string berupa head dari antrian

Algoritma

BuatAntrian(q) {antrian kosong}
BuatArray(daftarfile)
BuatArray(daftarsubdir)
MasukAntrian(q, root) {memasukkan directory pertama ke dalam
antrian}
|
```

```

while not AntrianKosong(q) do
    queuehead = dequeue(q) {dequeue elemen pertama queue menjadi
    queuehead}
    daftarfile = getFiles(queuehead) {mendapatkan semua file
    dalam queuehead}
    daftarsubdir = getDirectories (queuehead)

    for tiap subdir dalam daftarsubdir
        MasukAntrian(q,subdir)      {memasukkan subdir dalam
        antrian}
        BuatSisi(queuehead, subdir) {membangkitkan sisi antara
        queuehead dan subdir}

    endfor

    for tiap file dalam daftar file
        BuatSisi(queuehead, file)

        if(nama file sama dengan file yang dicari)
            colorgraph() {mewarnai rute menuju file hijau}
            if(not SearchAll)   {berhenti setelah ketemu}
                showAllInQueue() {menampilkan semua simpul
                dalam queue}

            return          {keluar dari prosedur}
            endif
        endif
    endfor
endwhile

```

B. Pseudocode Algoritma DFS

```

procedure DFS (input root : string, filename : string, SearchAll : bool)

{
    Melakukan traversal pada directory root menggunakan algoritma
    pencarian DFS

    Masukan : root adalah directory awal mulai pencarian, filename
    adalah nama file yang dicari, SearchAll adalah penanda apakah pengguna
    ingin mencari semua kemunculan file
    dalam directory tersebut

    Keluaran : graph hasil traversal dimunculkan ke layar dengan rute ke
    file yang dicari
    Berwarna hijau, rute yang dikunjungi namun tidak memiliki file
    merah, dan rute yang
    Sudah berada di queue namun tidak sempat dicek diwarnai hitam
}

```

```

Deklarasi
q : antrian directory
daftarfile, daftarsubdir : array berisi string
queuehead : string berupa head dari antrian

Algoritma

BuatArray(daftarfile)
BuatArray(daftarsubdir)
daftarfile = getFiles(root)
dafat subdir = getSubDir(root)

for tiap subdir dalam daftarsubdir

    BuatSisi(root,subdir)
    DFS(subdir, filename, searchAll) {rekursif}

endfor

for tiap file dalam daftarfile

    BuatSisi(root,file)
    if(nama file sama dengan yang dicari)
        if(not SearchAll)
            stopAllAndShow {memberhentikan seluruh rekursif
                            dan menampilkan semua simpul
                            dalam antrian rekursif}
            return
        endif
    endif
endfor

```

4.3 Struktur Data Program

Program folder crawler buatan kami menggunakan desain satu kelas C# yang bernama FolderCrawlerForm dengan namespace FolderCrawler. Alasan mengapa kami mengintegrasikan seluruh fungsi pada satu kelas agar mempermudah *passing message* antar method kelas. Kelas FolderCrawlerForm berisi:

a. Atribut

Kelas ini berisi atribut atribut sebagai berikut:

- string filename: digunakan untuk menampung nama file yang dicari.
- string root: digunakan untuk menampung nama folder yang menjadi akar pencarian.

- bool findAll (false): digunakan untuk menampung input pelanggan apakah cari semua kemungkinan file atau tidak
- string methodUsed: digunakan untuk menampung input pelanggan yang menjadi metode yang digunakan (BFS/DFS)
- bool DFSAlive (true): digunakan sebagai *flag* bagi fungsi rekursi DFS.
- double timeTaken: digunakan untuk menampung waktu eksekusi algoritma.
- List<string> foundLinks: digunakan untuk menampung hyperlink dari folder yang berisi file yang dicari.

b. Methods:

- public void wait(int milliseconds): digunakan untuk membuat jeda antar iterasi search agar kondisi mutual exclusion tercapai.
- public void colorGraph(string entry, Microsoft.Msagl.Drawing.Graph graph): digunakan untuk mewarnai jalan menuju simpul solusi dengan warna hijau.
- public void BFS(string root, string fileName, bool SearchAll): digunakan untuk mengeksekusi pencarian dengan metode Breadth First Search
- public void DFS(Microsoft.Msagl.GraphViewerGdi.GViewer viewer, Microsoft.Msagl.Drawing.Graph graph, string root, string fileName, bool SearchAll): digunakan untuk mengeksekusi pencarian dengan metode Depth First Search secara rekursif
- private void BFSRadioButton_CheckedChanged(object sender, EventArgs e): Digunakan sebagai event handler ketika radio button BFS dipilih.
- private void DFSButton_CheckedChanged(object sender, EventArgs e): Digunakan sebagai event handler ketika radio button DFS dipilih.
- private void button1_Click(object sender, EventArgs e): Digunakan sebagai event handler ketika button search diklik (mengeksekusi search)
- private void chooseFolder_Click(object sender, EventArgs e): Digunakan sebagai event handler ketika button select folder diklik (menampilkan dialog select folder)
- private void inputFilename_TextChanged(object sender, EventArgs e): Digunakan sebagai event handler ketika nama file yang ingin dicari berubah.

- private void allOccurence_CheckedChanged(object sender, EventArgs e): Digunakan sebagai event handler ketika Find all occurence diisi maupun tidak.
- private void hyperlinkListBox_SelectedIndexChanged(object sender, EventArgs e): Digunakan sebagai event handler ketika anggota hyperlinkListBox diklik (membuka hyperlink).

c. Constructor

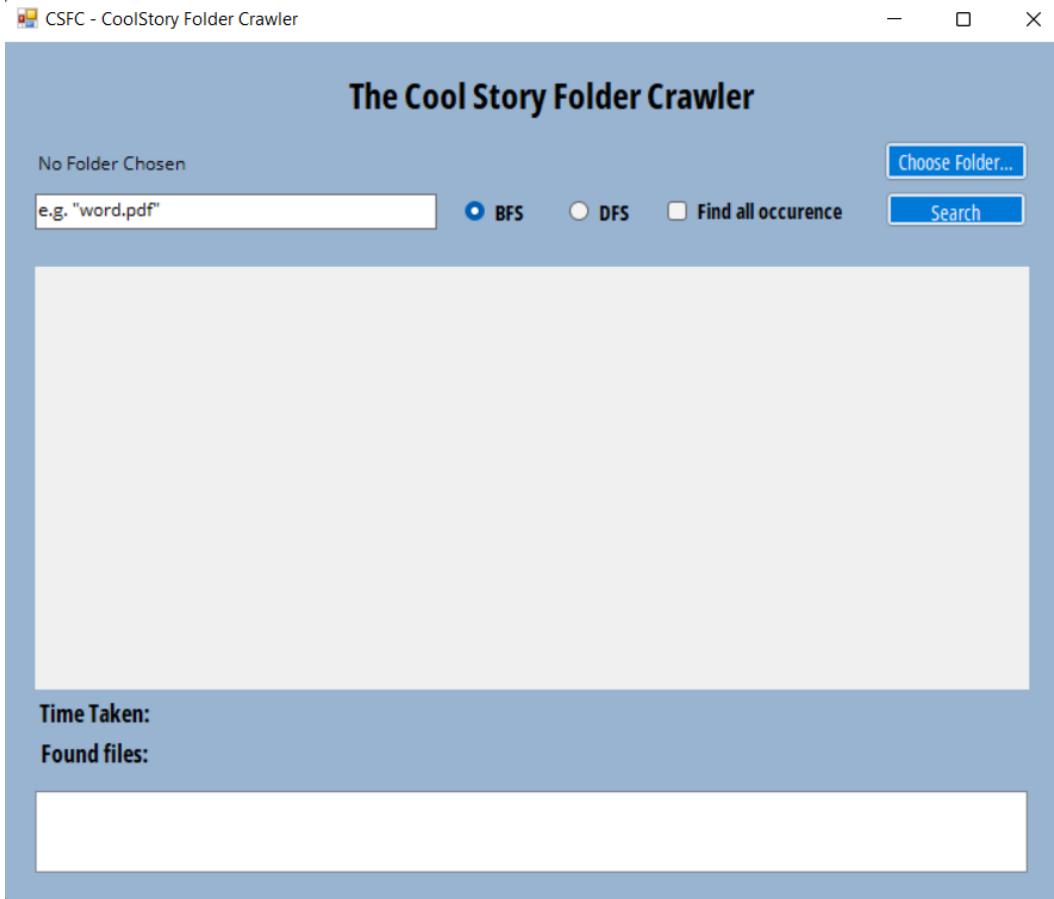
- public FolderCrawlerForm(): Digunakan untuk menginisialisasi komponen form.

Selain kelas utama pada Form1.cs dan design GUI pada Form1.Designer.cs, entrypoint dari program ini sendiri ada pada Program.cs yang menginisialisasi objek FolderCrawlerForm yang berada pada Form1.cs.

4.4 Tata Cara Penggunaan Program

Program dapat langsung dijalankan dengan file “.exe”-nya ataupun dapat di-build terlebih dahulu di Visual Studio.

- a. Untuk menjalankan program, dapat di-run secara langsung melalui file FolderCrawler.exe yang terdapat pada direktori “./bin/Release”.
- b. Jika ingin melakukan build, buka file solution FolderCrawler.sln pada “./src” dengan menggunakan Visual Studio dan tekan F5 ataupun CTRL + F5 untuk melakukan build. Hasil dari build tersebut akan langsung muncul dan program dapat langsung digunakan.

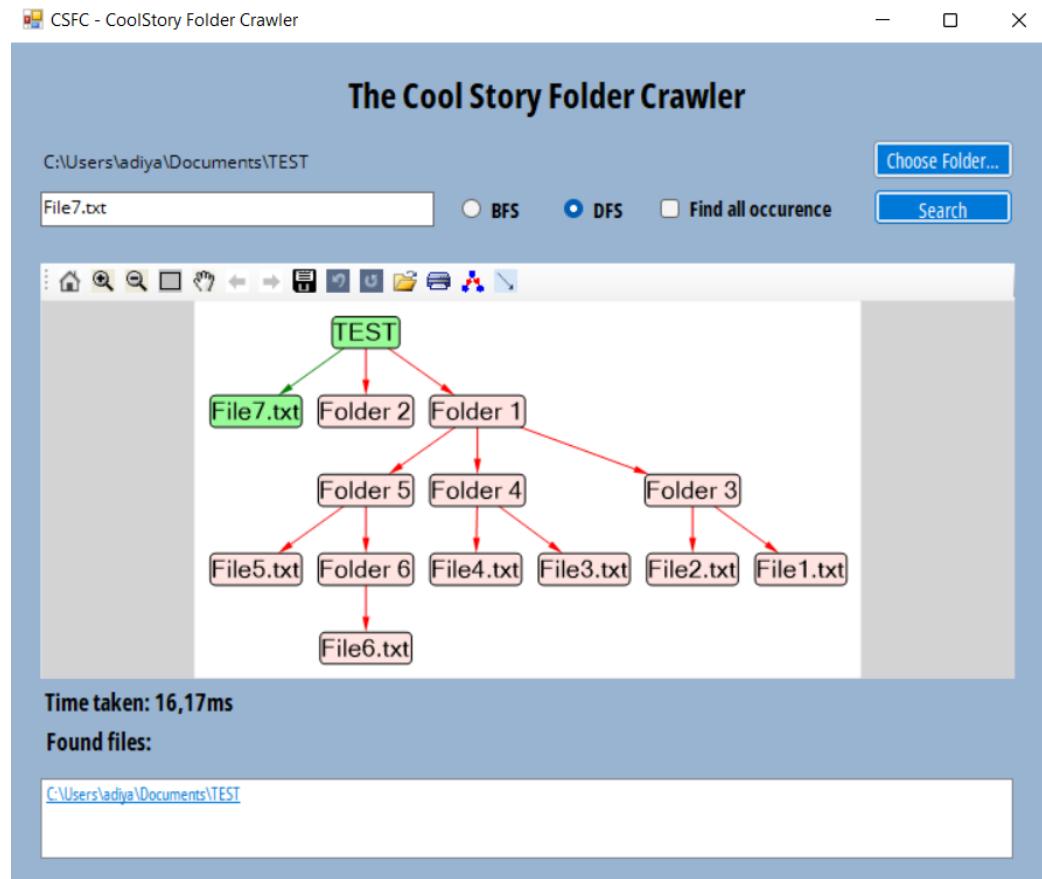


Gambar 4.4.1 User Interface Folder Crawler

Untuk melakukan pencarian *file* dengan menggunakan FolderCrawler.exe dapat dilakukan dengan cara sebagai berikut.

1. Pilih *root folder* dengan memencet button “Choose Folder” dan silahkan arahkan kepada *folder* sesuai yang diinginkan.
2. Ketik nama *file* yang ingin dicari pada *folder* yang telah dipilih sebelumnya
3. Pilih metode pencarian *file* yang diinginkan dengan cara memencet salah satu dari tombol BFS ataupun DFS.
4. Sebagai opsi tambahan untuk mencari semua *file* yang terdapat pada *root folder*; dapat memencet box “Find all occurrence”.
5. Pencet tombol “Search” jika langkah 1 sampai 4 telah dilakukan. Visualisasi dari pencarian *file* akan tampil pada label kosong. Waktu algoritma berjalan juga akan muncul

beserta *hyperlink* yang bisa di-klik untuk langsung membuka *folder* ditemukannya *file* yang dicari.



Gambar 4.4.2 Contoh Program Selesai

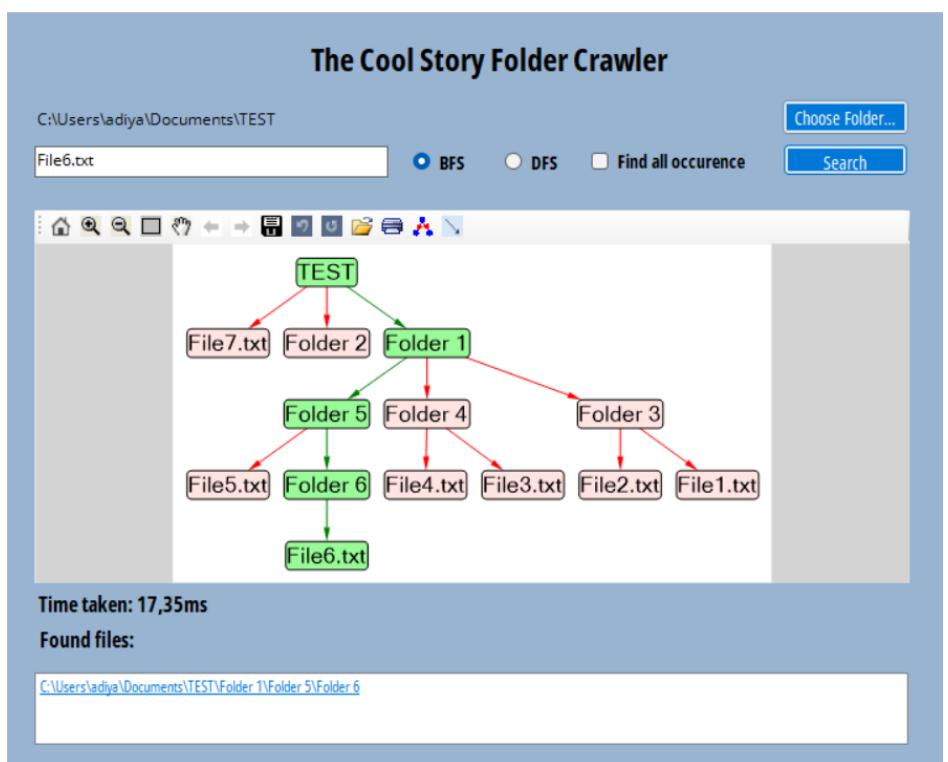
4.5 Pengujian dan Hasil

- Folder TEST dengan File Target “File6.txt”

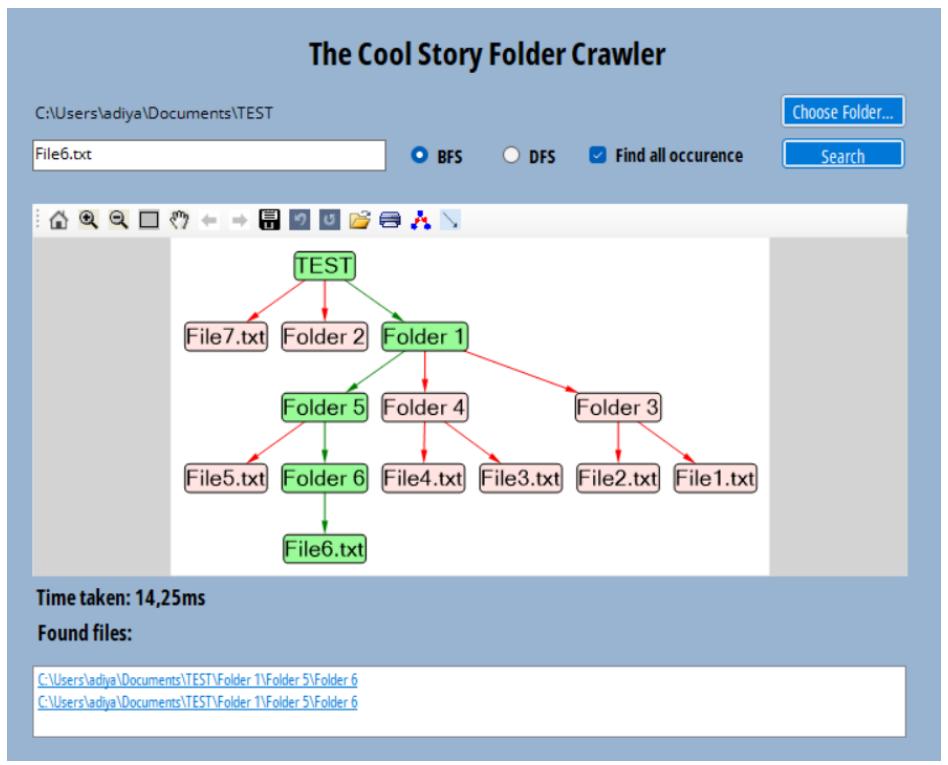
Tabel 4.5.1 Pengujian 1

<i>Hierarki Folder</i>	
<pre>graph TD; TEST[TEST] --> File7[File7.txt]; TEST --> Folder2[Folder 2]; TEST --> Folder1[Folder 1]; Folder1 --> Folder5[Folder 5]; Folder1 --> Folder4[Folder 4]; Folder1 --> Folder3[Folder 3]; Folder5 --> File5[File5.txt]; Folder5 --> Folder6[Folder 6]; Folder4 --> File4[File4.txt]; Folder4 --> File3[File3.txt]; Folder3 --> File2[File2.txt]; Folder3 --> File1[File1.txt]; Folder6 --> File6[File6.txt]</pre>	
File Target	File6.txt
Metode	<i>Hasil Uji</i>

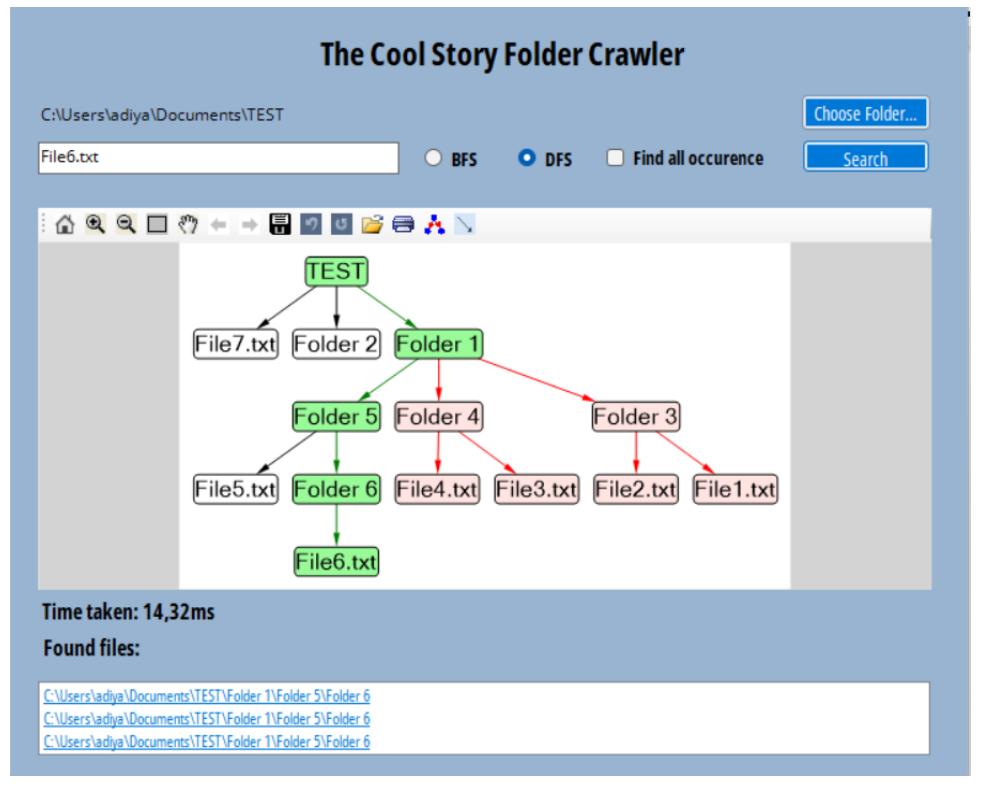
BFS



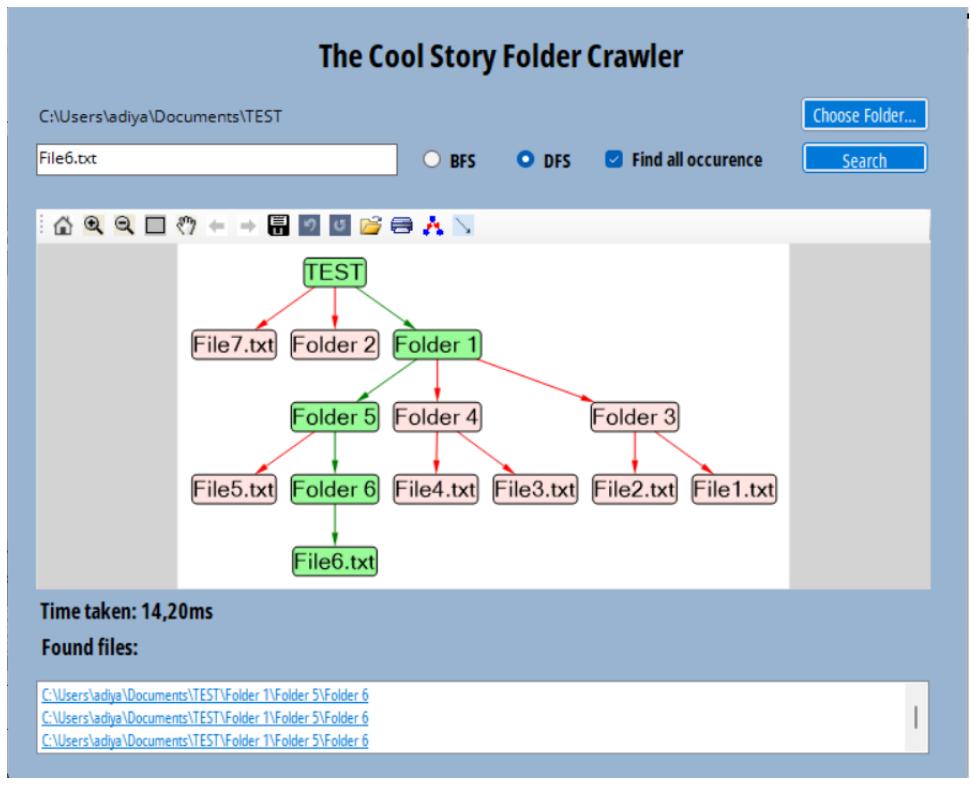
BFS All Occurrence



DFS



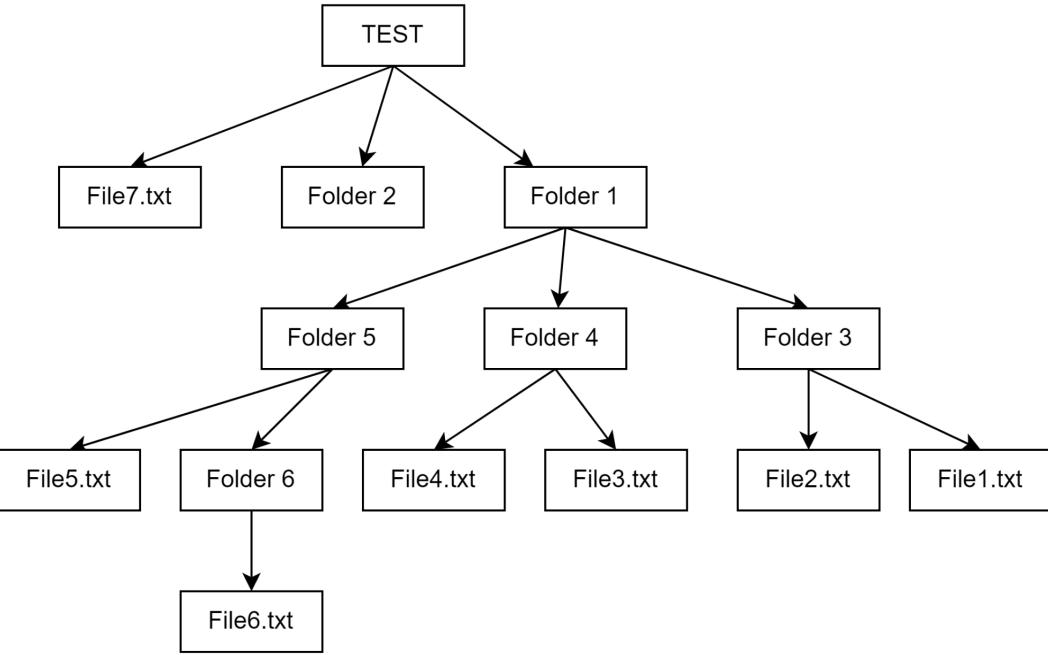
DFS All Occurrence



b. Folder TEST File Target “File7.txt”

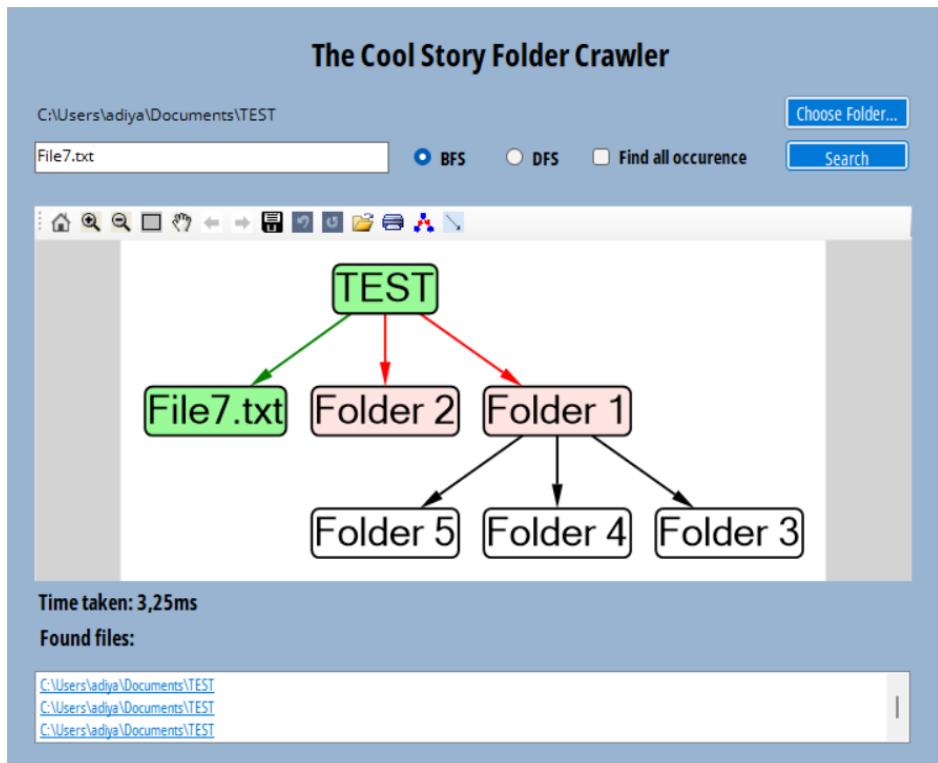
Tabel 4.5.2 Pengujian 2

Hierarki Folder

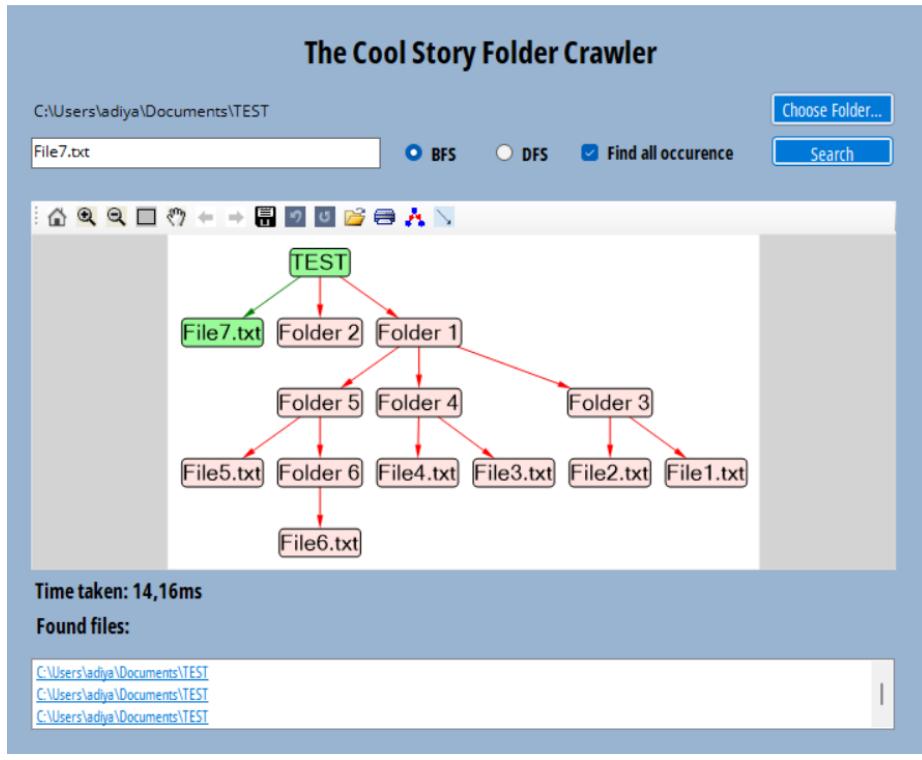


File Target	File7.txt
Metode	<i>Hasil Uji</i>

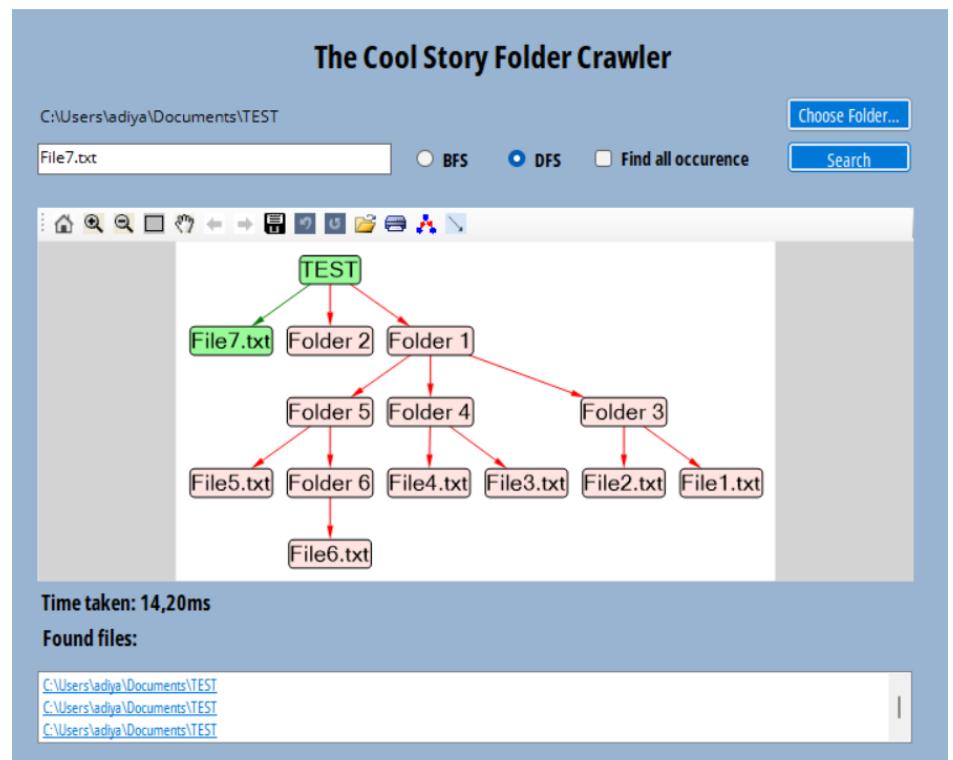
BFS



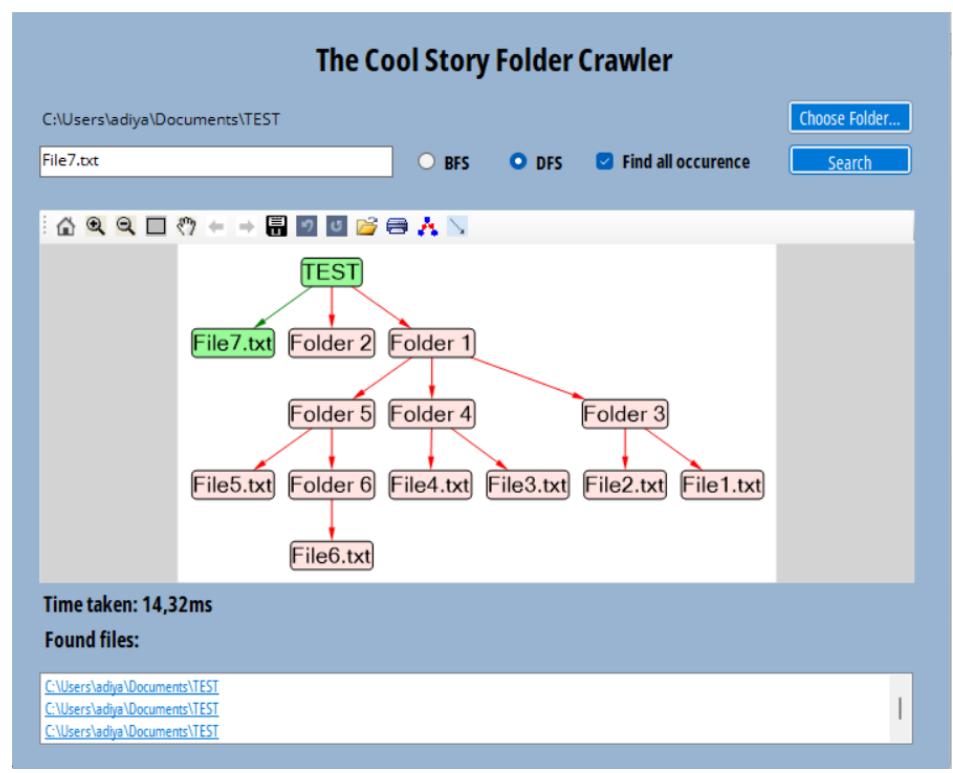
BFS All Occurrence



DFS



DFS All Occurrence

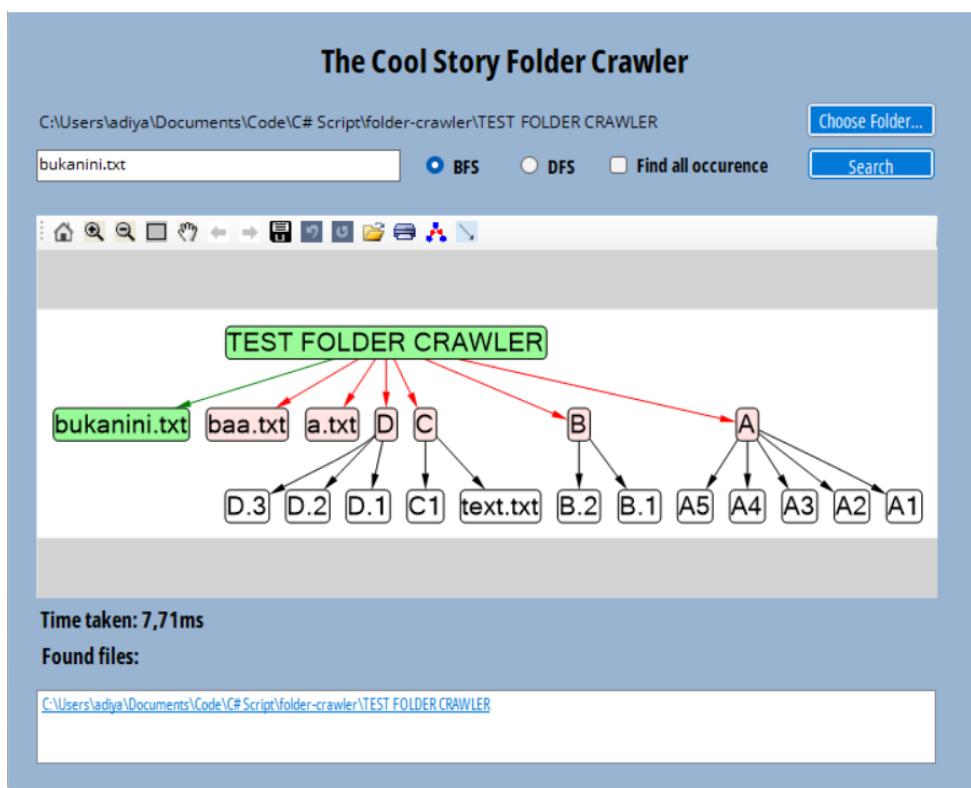


c. Folder TEST FOLDER CRAWLER File Target “bukanini.txt”

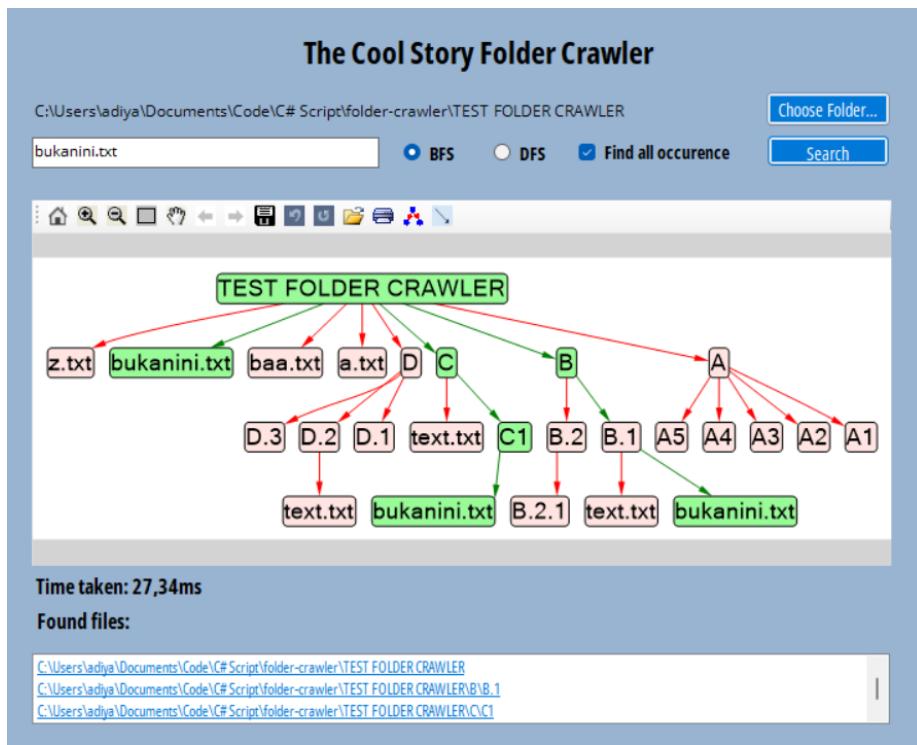
Tabel 4.5.3 Pengujian 3

<i>Hierarki Folder</i>	
<pre>graph TD; Root[TEST FOLDER CRAWLER] --> ztxt[z.txt]; Root --> bukaninixt1[bukanini.txt]; Root --> baatxt[baa.txt]; Root --> atxt[a.txt]; Root --> D[D]; Root --> C[C]; Root --> B[B]; Root --> A[A]; D --> D3[D.3]; D --> D2[D.2]; D --> D1[D.1]; D --> texttxt1[text.txt]; C --> C1[C1]; B --> B2[B.2]; B --> B1[B.1]; A --> A5[A5]; A --> A4[A4]; A --> A3[A3]; A --> A2[A2]; A --> A1[A1]; D2 --> texttxt2[text.txt]; C1 --> bukaninixt2[bukanini.txt]; B2 --> B21[B.2.1]; B1 --> texttxt3[text.txt]; B1 --> bukaninixt4[bukanini.txt]</pre>	
File Target	bukanini.txt
Metode	<i>Hasil Uji</i>

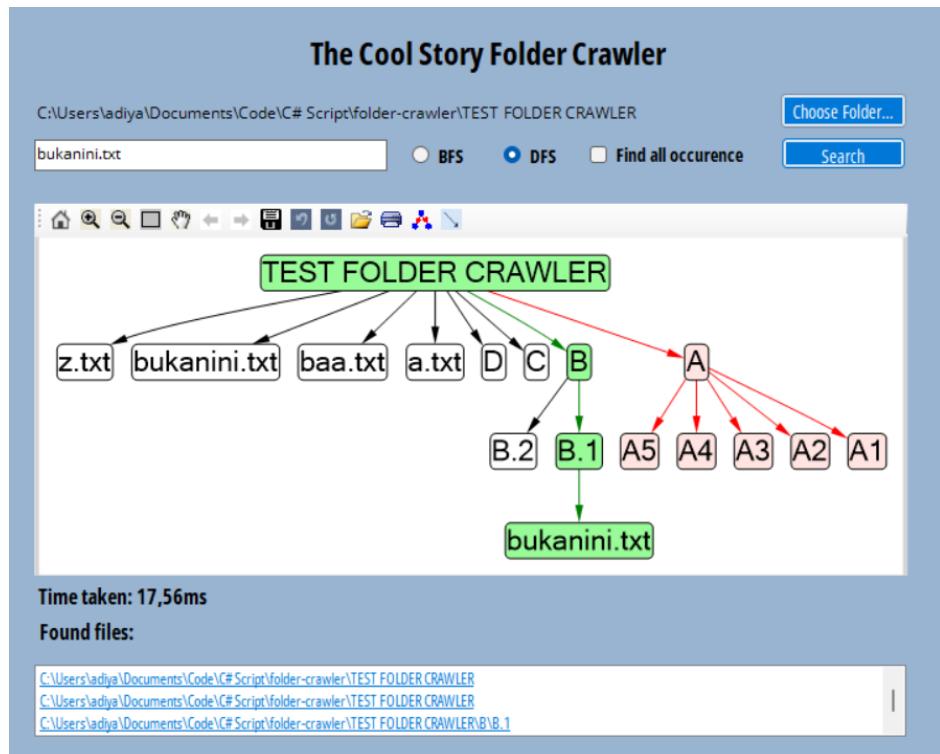
BFS



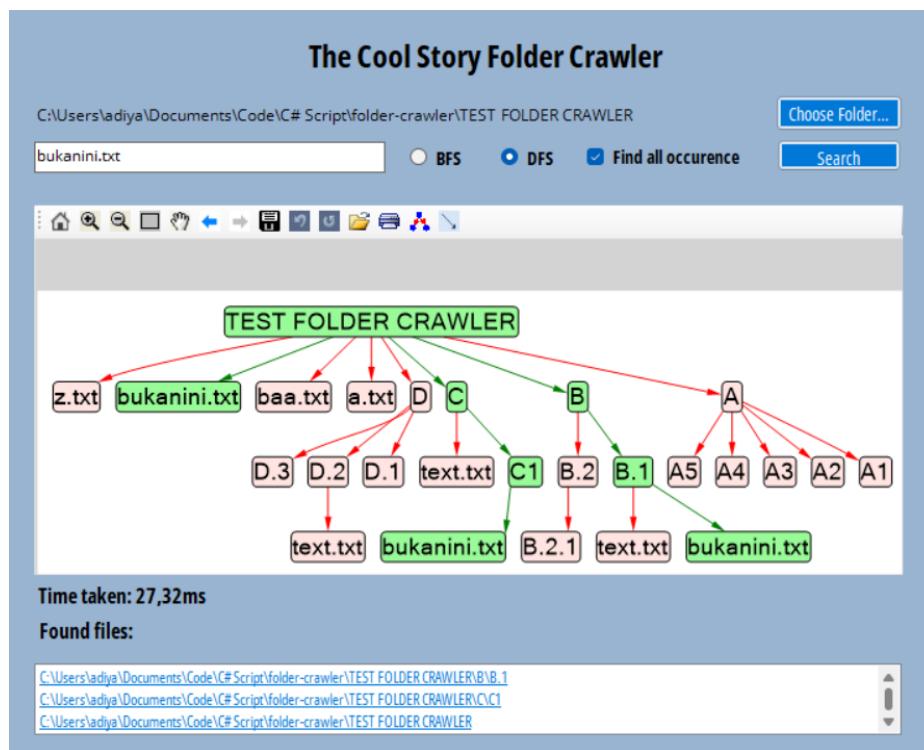
BFS All Occurrence



DFS



DFS All Occurrence

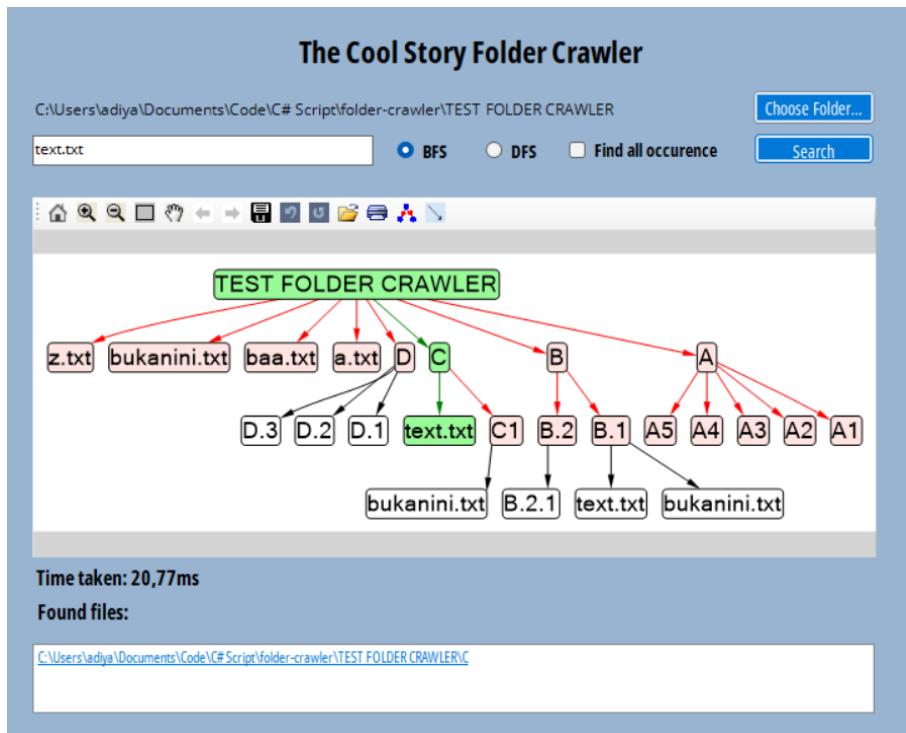


d. Folder TEST FOLDER CRAWLER File Target “text.txt”

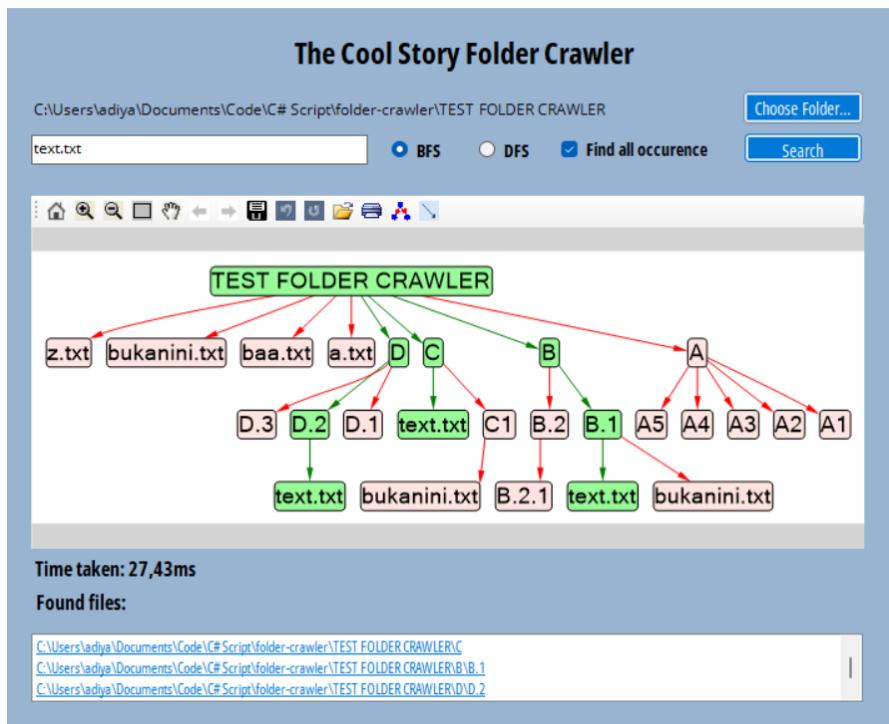
Tabel 4.5.4 Pengujian 4

<i>Hierarki Folder</i>	
	<pre>graph TD; Root[TEST FOLDER CRAWLER] --> ztxt[z.txt]; Root --> bukaninixtxt[bukanini.txt]; Root --> baaxtxt[baa.txt]; Root --> atxt[a.txt]; Root --> D[D]; Root --> C[C]; Root --> B[B]; Root --> A[A]; D --> D3[D.3]; D --> D2[D.2]; D --> D1[D.1]; D --> texttxt{text.txt}; C --> C1[C1]; B --> B2[B.2]; B --> B1[B.1]; B1 --> B21[B.2.1]; B1 --> texttxt2{text.txt}; B1 --> bukaninixtxt2[bukanini.txt]; A --> A5[A5]; A --> A4[A4]; A --> A3[A3]; A --> A2[A2]; A --> A1[A1]; A1 --> texttxt3{text.txt}</pre>
File Target	text.txt
Metode	Hasil Uji

BFS



BFS All Occurrence



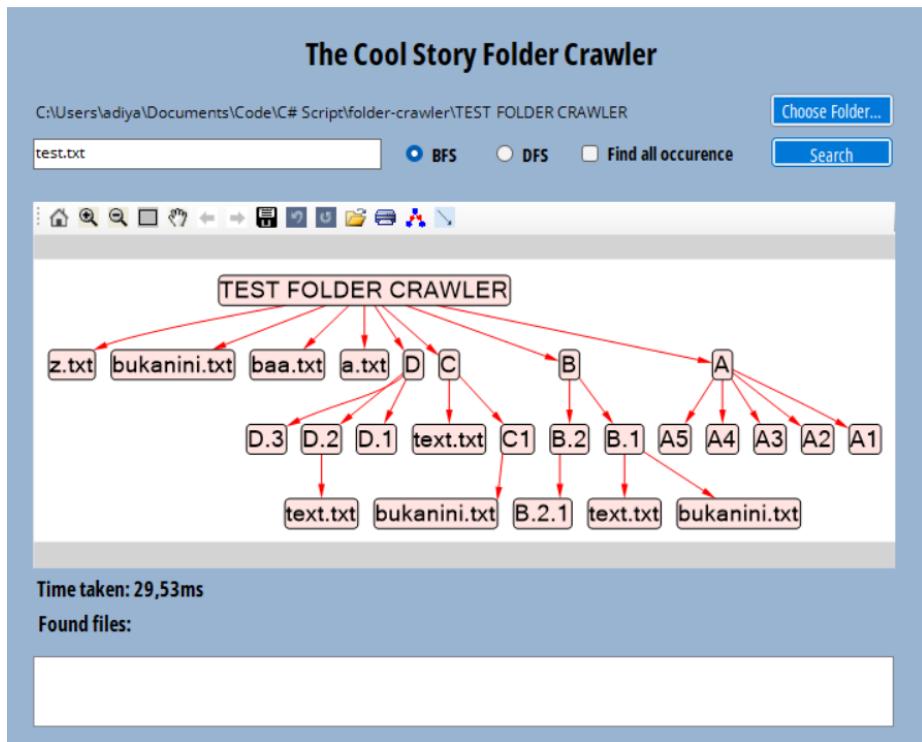
DFS	<p>The Cool Story Folder Crawler</p> <p>C:\Users\adiya\Documents\Code\C# Script\folder-crawler\TEST FOLDER CRAWLER</p> <p><input type="text" value="text.txt"/> <input checked="" type="radio"/> BFS <input checked="" type="radio"/> DFS <input type="checkbox"/> Find all occurrence <input type="button" value="Search"/></p> <p>Time taken: 18,61ms</p> <p>Found files:</p> <ul style="list-style-type: none"> C:\Users\adiya\Documents\Code\C# Script\folder-crawler\TEST FOLDER CRAWLER\b\b.1 C:\Users\adiya\Documents\Code\C# Script\folder-crawler\TEST FOLDER CRAWLER\d\d.2 C:\Users\adiya\Documents\Code\C# Script\folder-crawler\TEST FOLDER CRAWLER\b\b.1
DFS All Occurrence	<p>The Cool Story Folder Crawler</p> <p>C:\Users\adiya\Documents\Code\C# Script\folder-crawler\TEST FOLDER CRAWLER</p> <p><input type="text" value="text.txt"/> <input checked="" type="radio"/> BFS <input checked="" type="radio"/> DFS <input checked="" type="checkbox"/> Find all occurrence <input type="button" value="Search"/></p> <p>Time taken: 27,52ms</p> <p>Found files:</p> <ul style="list-style-type: none"> C:\Users\adiya\Documents\Code\C# Script\folder-crawler\TEST FOLDER CRAWLER\b\b.1 C:\Users\adiya\Documents\Code\C# Script\folder-crawler\TEST FOLDER CRAWLER\b\b.1 C:\Users\adiya\Documents\Code\C# Script\folder-crawler\TEST FOLDER CRAWLER\d\d.2

e. Folder TEST FOLDER CRAWLER File Target “test.txt”

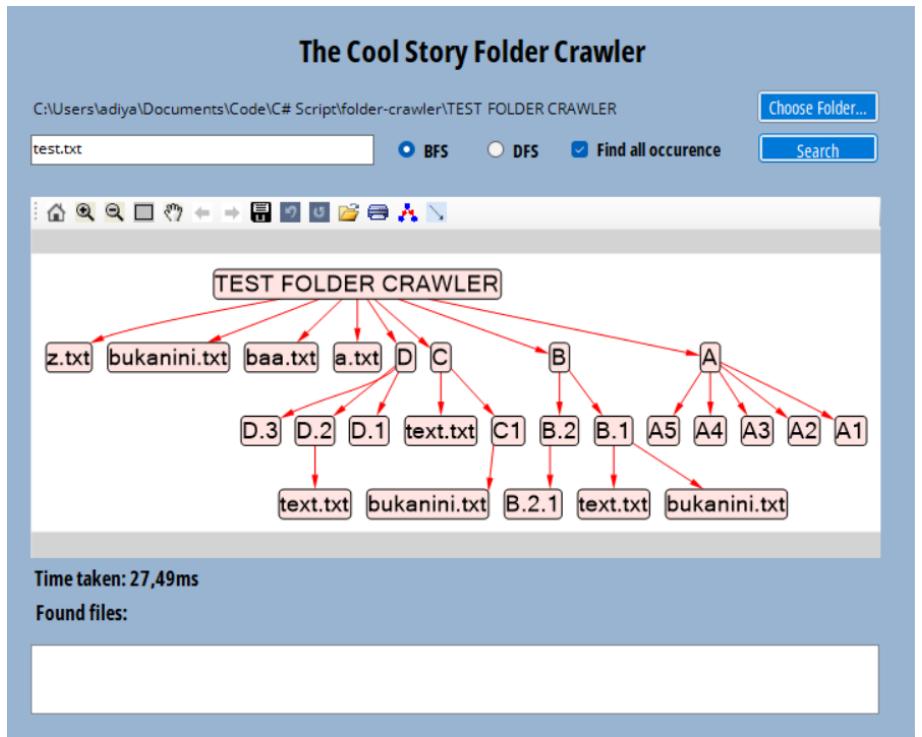
Tabel 4.5.5 Pengujian 5

<i>Hierarki Folder</i>	
<pre> graph TD A[TEST FOLDER CRAWLER] --> ztxt[z.txt] A --> bukaninixt1[bukanini.txt] A --> baa[baa.txt] A --> atxt[a.txt] A --> D[D] A --> C[C] A --> B[B] A --> A[A] D --> D3[D.3] D --> D2[D.2] D --> D1[D.1] D --> texttxt1[text.txt] C --> C1[C1] B --> B2[B.2] B --> B1[B.1] A --> A5[A5] A --> A4[A4] A --> A3[A3] A --> A2[A2] A --> A1[A1] D3 --> texttxt2[text.txt] C1 --> bukaninixt2[bukanini.txt] B2 --> B21[B.2.1] B1 --> texttxt3[text.txt] B1 --> bukaninixt4[bukanini.txt] </pre>	
File Target	test.txt
Metode	<i>Hasil Uji</i>

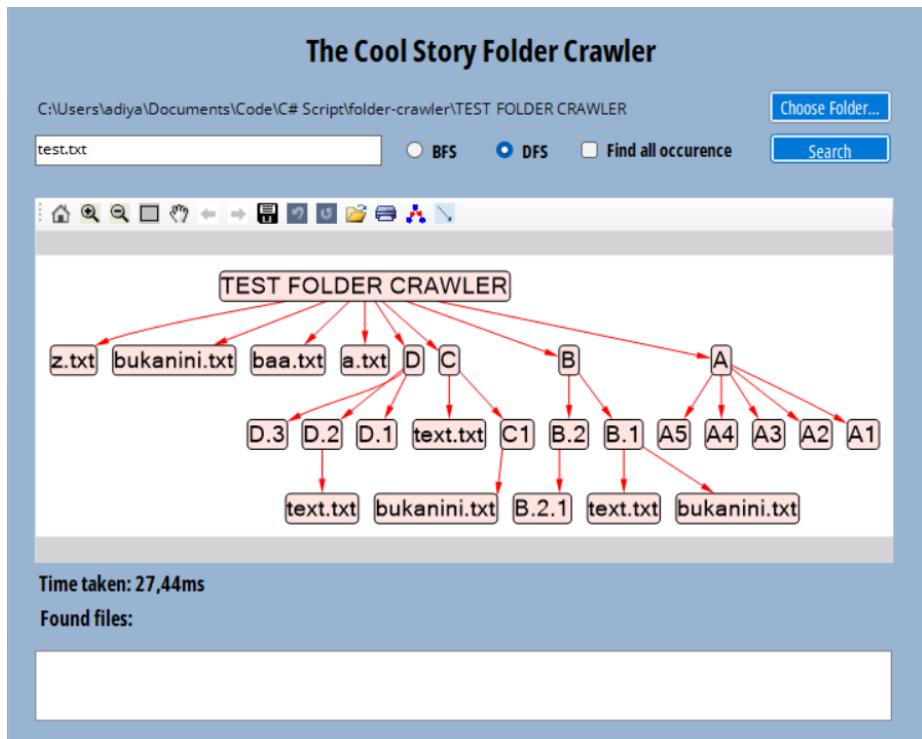
BFS



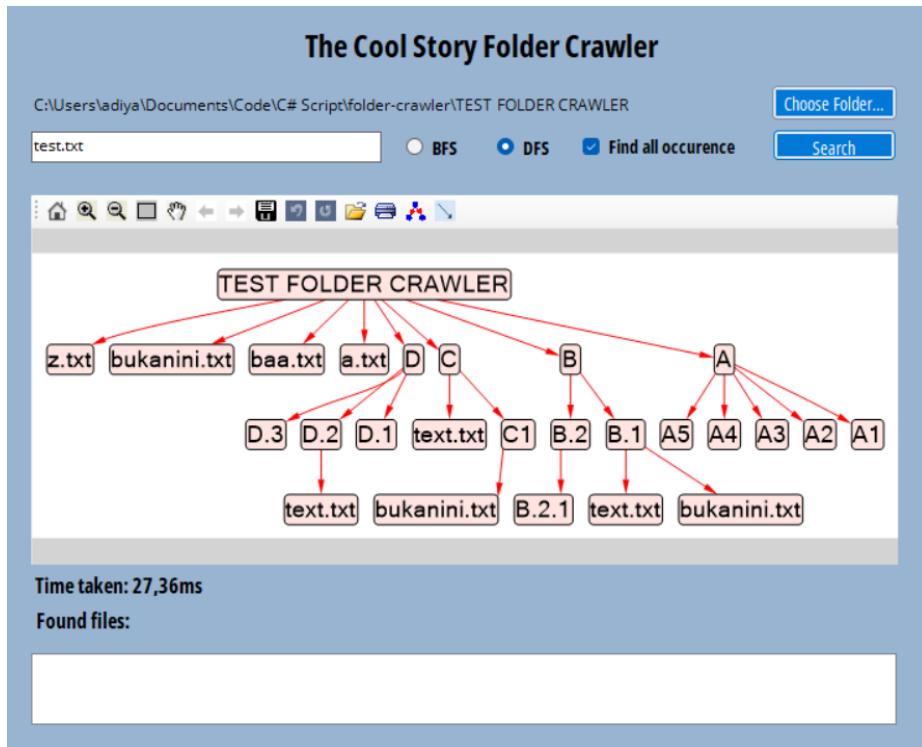
BFS All Occurrence



DFS



DFS All Occurrence



4.6 Analisis

Terdapat 5 pengujian yang dilakukan terhadap program ini dengan menggunakan 2 *root folder* dan 5 *file* yang berbeda tiap pengujinya. Pengujian 1 dan 2 menggunakan *root folder* “TEST” dan 2 *file* yaitu “File6.txt” dan “File7.txt”. Pengujian ini dilakukan untuk memperlihatkan *behaviour* yang berbeda antara pencarian dengan BFS dan DFS (bukan *all occurrence*). Pengujian 1 pencarian terhadap “File6.txt” dan *file* tersebut diletakkan pada *depth* yang paling dalam dari *tree*, sehingga pencarian dengan DFS akan lebih dahulu menemukan “File6.txt” karena DFS merupakan pencarian secara mendalam, sedangkan BFS menemukan “File6.txt” di paling akhir. Hal ini terbukti dengan waktu yang dibutuhkan lebih cepat pada algoritma DFS. Sedangkan, untuk Pengujian 2 “File7.txt” diletakkan pada *root folder* sehingga pencarian dengan BFS akan langsung menemukan *file target*-nya, hal ini juga terbukti dengan waktu yang dibutuhkan jauh lebih cepat untuk algoritma BFS.

Pengujian 3 dan 4 dilakukan untuk memperlihatkan *behaviour* pencarian secara BFS dan DFS terhadap beberapa *file target* yang sama. Pada pengujian 3, “bukanini.txt” yang ditemukan lebih dahulu oleh algoritma BFS adalah di *root folder*-nya, sedangkan “bukanini.txt” yang ditemukan oleh algoritma DFS terdapat pada folder “B.1” yang merupakan *depth* paling dalam. Untuk pengujian 4, “text.txt” yang ditemukan lebih dahulu oleh algoritma BFS terdapat pada folder “C”, sedangkan “text.txt” ditemukan oleh algoritma DFS pada folder “B.1”. Perilaku pada pengujian dari 1 sampai 4 sudah sesuai dengan perilaku yang seharusnya dilakukan oleh masing-masing algoritma tersebut. Untuk algoritma BFS dan DFS *all occurrence* akan menghasilkan *tree* yang sama karena dilakukan pencarian secara menyeluruh sehingga semua *internal nodes* dan *leaf nodes* akan semuanya dilakukan pengecekan. Untuk waktu yang dibutuhkan algoritma BFS dan DFS *all occurrence* keduanya sangat mirip sehingga tidak bisa ditentukan mana algoritma yang lebih cepat karena kedua algoritma sangat bergantung terhadap situasi.

Pengujian terakhir yaitu pengujian 5 dilakukan pencarian terhadap file yang tidak ada di dalam *root folder* tersebut. Hasil yang diberikan oleh tiap algoritma semuanya sama dan sudah sesuai dengan hasil yang diinginkan. Waktu yang dibutuhkan untuk mengunjungi tiap *nodes* juga relatif mirip antara kedua algoritma. Hal ini mungkin disebabkan oleh penggunaan fungsi *wait()*

di tiap iterasi maupun rekursi, sehingga waktu total yang dibutuhkan akan bertambah secara *fixed* sesuai waktu *wait*-nya. Walaupun, pada akhir total waktu yang dibutuhkan dibagi dengan waktu *wait*, tetapi pada akhirnya cara seperti itu akan menghasilkan waktu tiap iterasi/rekursi menjadi 1 ms.

BAB 5

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Dari tugas besar 2 IF2211 Strategi Algoritma Semester 2 Tahun Ajaran 2021/2022 ini, kami berhasil membuat *desktop application* Folder Crawling. Program akhir dibuat dengan menggunakan GUI. Program akhir juga mampu melakukan pencarian dengan algoritma DFS ataupun BFS serta pencarian menyeluruh sekaligus menampilkan visualisasi bertahap pada pohon pencarian. Pencarian yang berhasil ditemukan juga akan menampilkan *hyperlink* yang jika dipencet akan menampilkan file explorer pada direktori ditemukannya *file* yang dicari.

5.2 Saran

Saran untuk kelompok kami di antaranya :

1. Pembagian tugas dilakukan dengan lebih baik lagi, agar alur bekerja lebih jelas.
2. Algoritma yang digunakan pada tugas besar ini masih dapat dikembangkan lebih lanjut lagi, baik dari segi format kode sampai efektivitas algoritma.
3. Kode program diberi komentar yang lebih jelas lagi. Agar kode yang digunakan lebih mudah dimengerti dan semua anggota bisa melakukan *debugging* pada algoritma yang bermasalah.

DAFTAR PUSTAKA

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

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

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf>

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag2.pdf>

<https://github.com/microsoft/automatic-graph-layout>

<https://docs.microsoft.com/en-us/dotnet/csharp/>

https://www.kirupa.com/developer/actionscript/depth_breadth_search.htm

<https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>