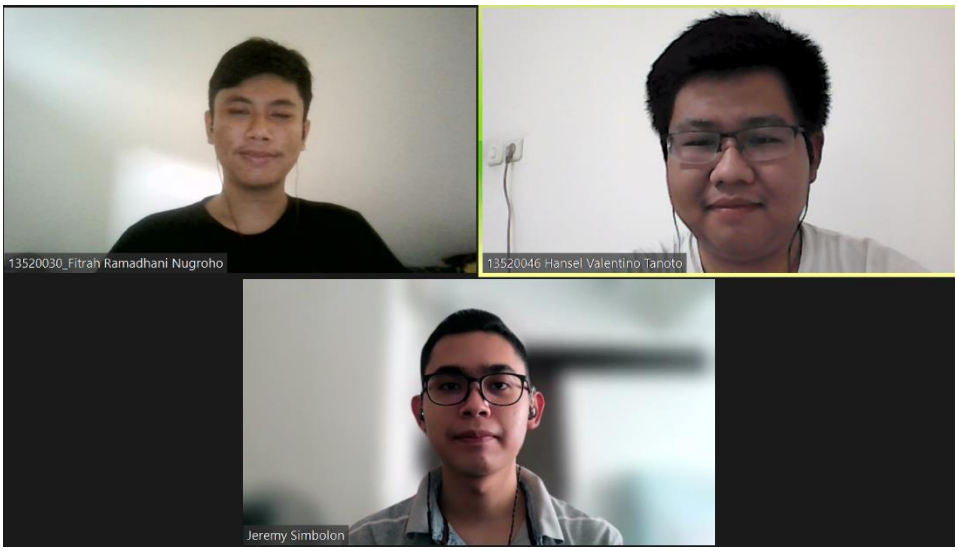
LAPORAN TUGAS BESAR 2

IF2211 STRATEGI ALGORITMA

Kelompok 24 – Pencarian Roti Pertama



Disusun oleh:

Fitrah Ramadhani Nugroho	13520030
Jeremy S.O.N. Simbolon	13520042
Hansel Valentino Tanoto	13520046

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

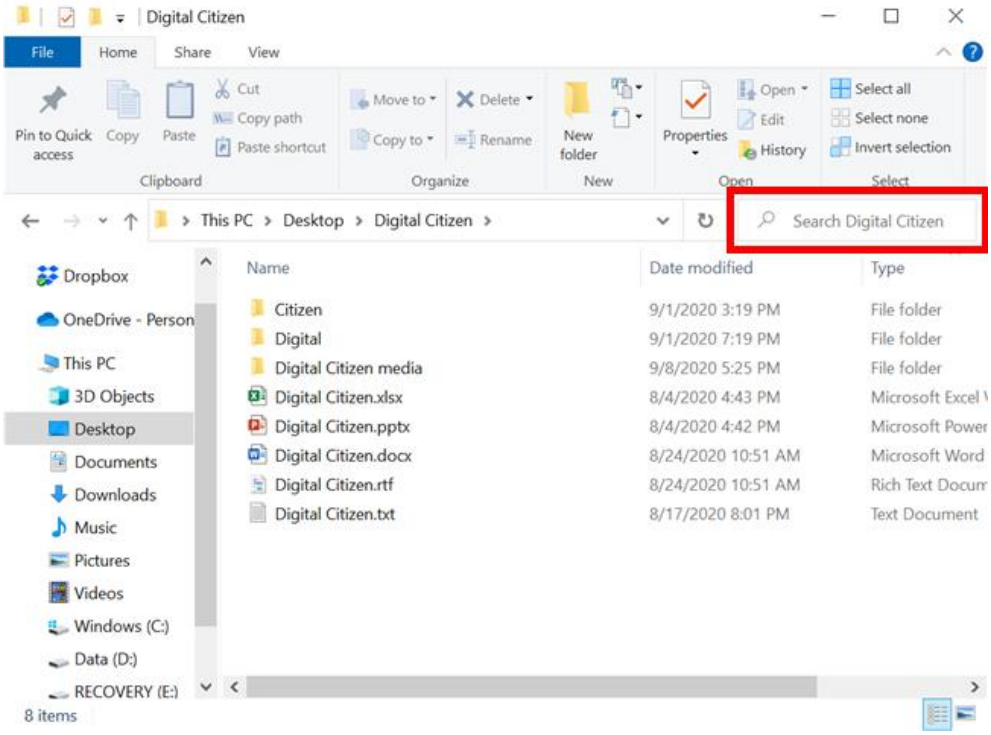
2022

DAFTAR ISI

BAB 1 – DESKRIPSI TUGAS

Latar Belakang

Pada saat kita ingin mencari *file* spesifik yang tersimpan pada komputer kita, seringkali *task* tersebut membutuhkan waktu yang lama apabila kita melakukannya secara manual. Bukan saja harus membuka beberapa *folder* hingga dapat mencapai *directory* yang diinginkan, kita bahkan dapat lupa di mana kita meletakkan *file* tersebut. Sebagai akibatnya, kita harus membuka berbagai *folder* secara satu persatu hingga kita menemukan *file* yang diinginkan. Hal ini pastinya akan sangat memakan waktu dan energi.



Gambar 1. Fitur Search pada Windows 10 File Explorer

(Sumber: https://www.digitalcitizen.life/wp-content/uploads/2020/10/explorer_search_10.png)

Meskipun demikian, kita tidak perlu cemas dalam menghadapi persoalan tersebut sekarang. Pasalnya, hampir seluruh sistem operasi sudah menyediakan fitur *search* yang dapat digunakan untuk mencari file yang kita inginkan. Kita cukup memasukkan *query* atau kata kunci pada kotak pencarian, dan komputer akan mencarikan seluruh *file* pada suatu *starting directory* (hingga seluruh *children*-nya) yang berkorespondensi terhadap *query* yang kita masukkan.

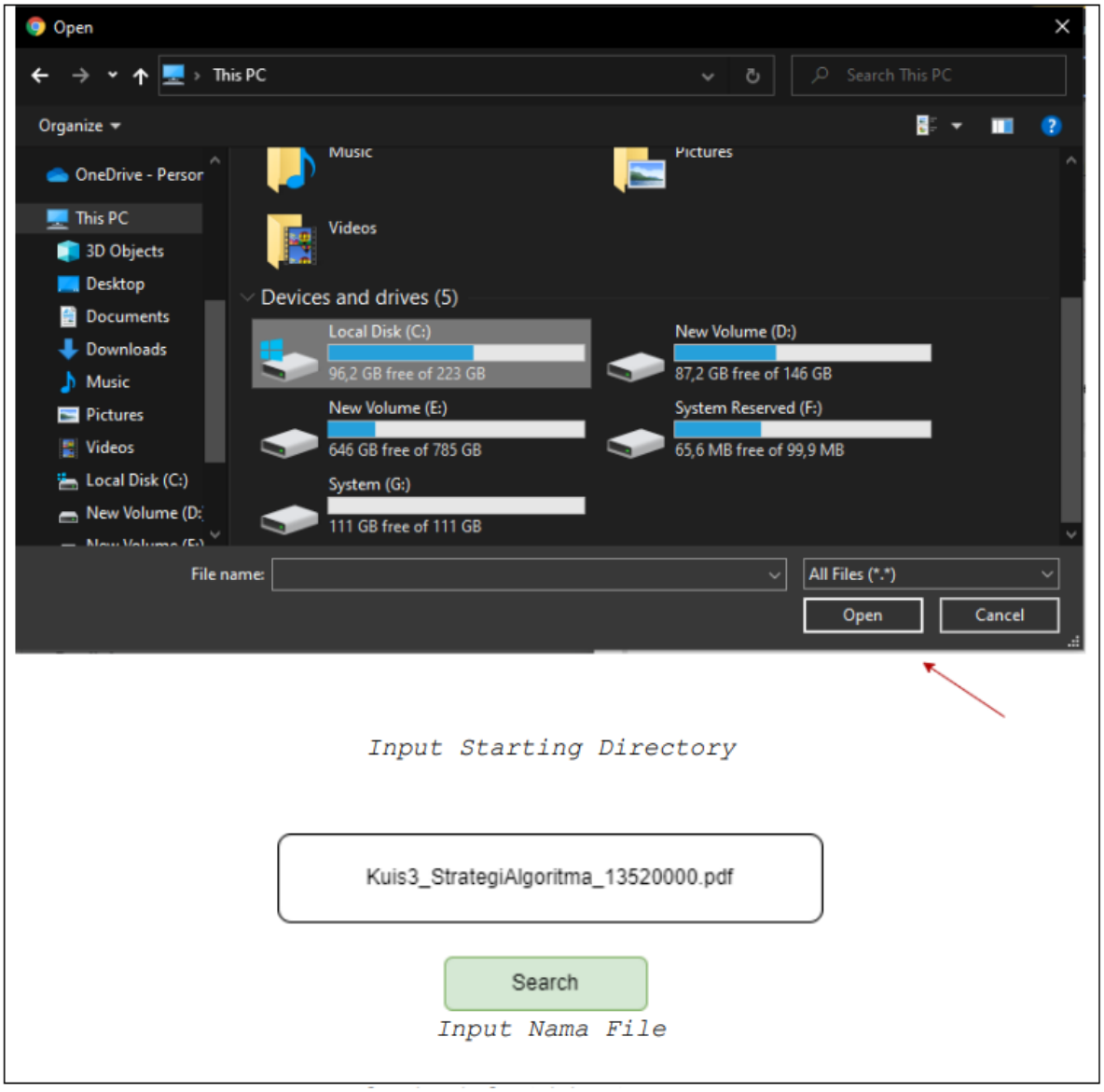
Fitur ini diimplementasikan dengan teknik *folder crawling*, di mana mesin komputer akan mulai mencari *file* yang sesuai dengan *query* mulai dari *starting directory* hingga seluruh *children* dari *starting directory* tersebut sampai satu *file* pertama/seluruh *file* ditemukan atau tidak ada *file* 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.

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.

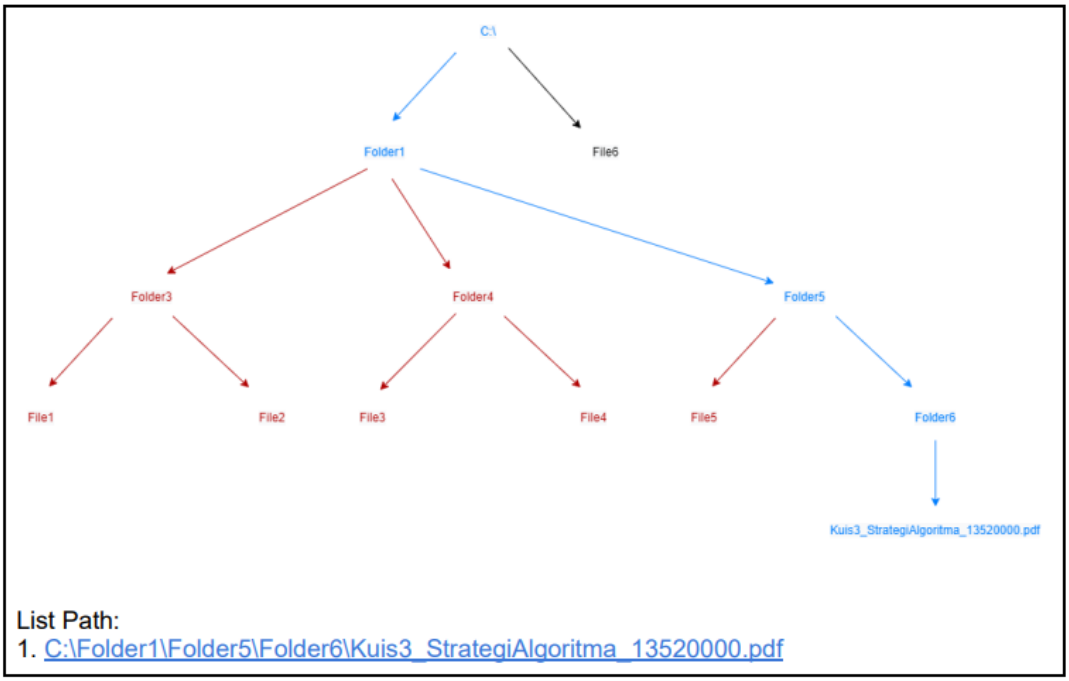
Contoh *Input* dan *Output* Program

Contoh masukan aplikasi:



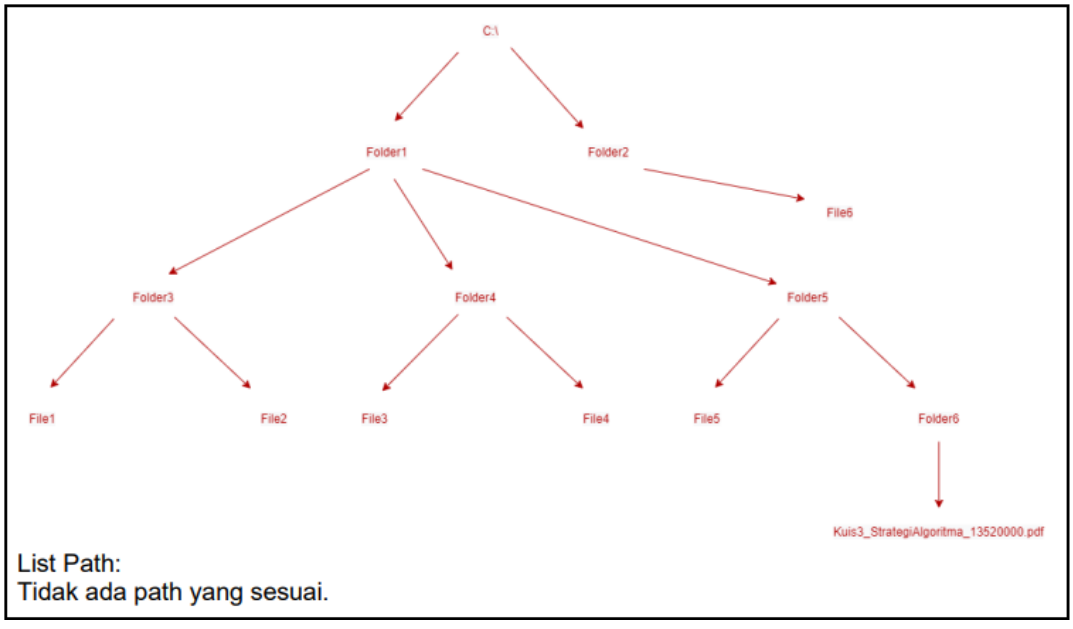
Gambar 2. Contoh input program

Contoh *output* aplikasi:



Gambar 3. Contoh output program

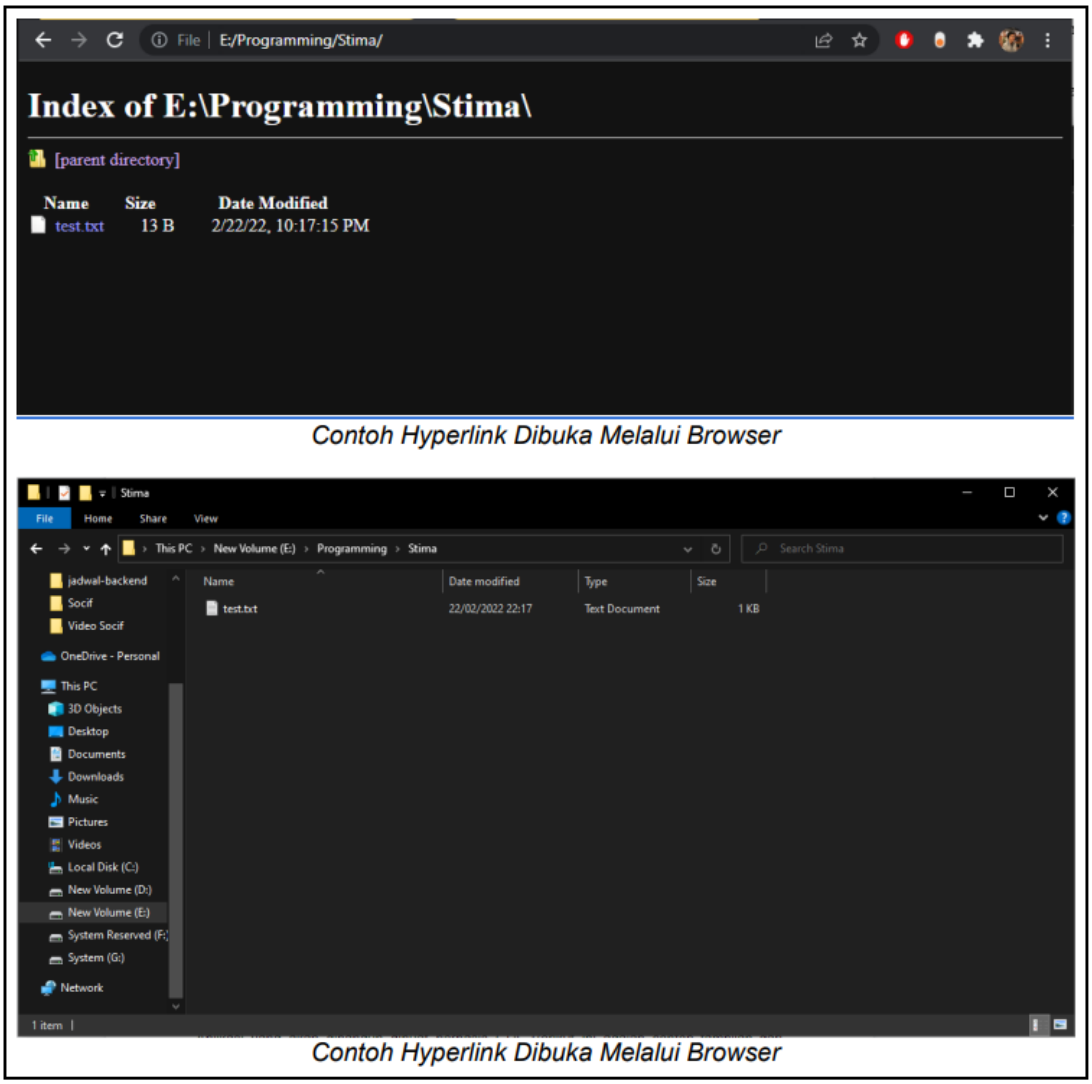
Misalnya pengguna ingin mengetahui langkah *folder crawling* untuk menemukan *file* Kuis3_StrategiAlgoritma_13520000.pdf. Maka, *path* pencarian DFS adalah sebagai berikut. C:\ → Folder1 → Folder3 → File1 → Folder3 → File2 → Folder3 → Folder1 → Folder4 → File3 → Folder4 → File4 → Folder4 → Folder1 →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



Gambar 4. 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.

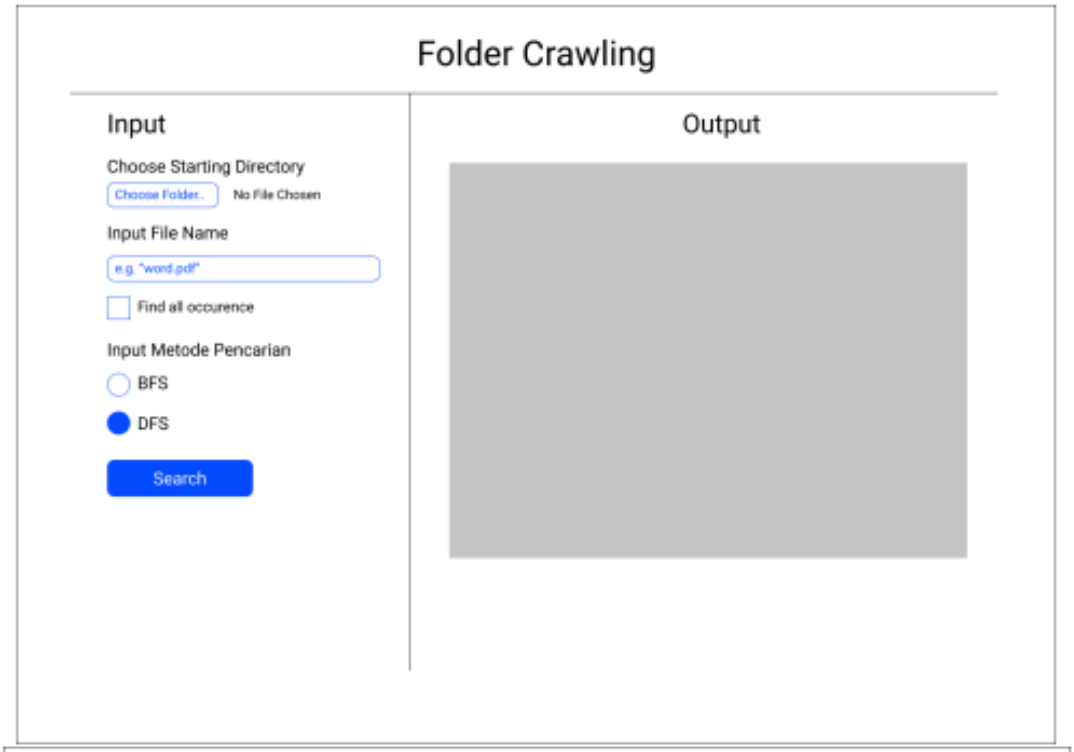
Contoh *Hyperlink* Pada *Path*

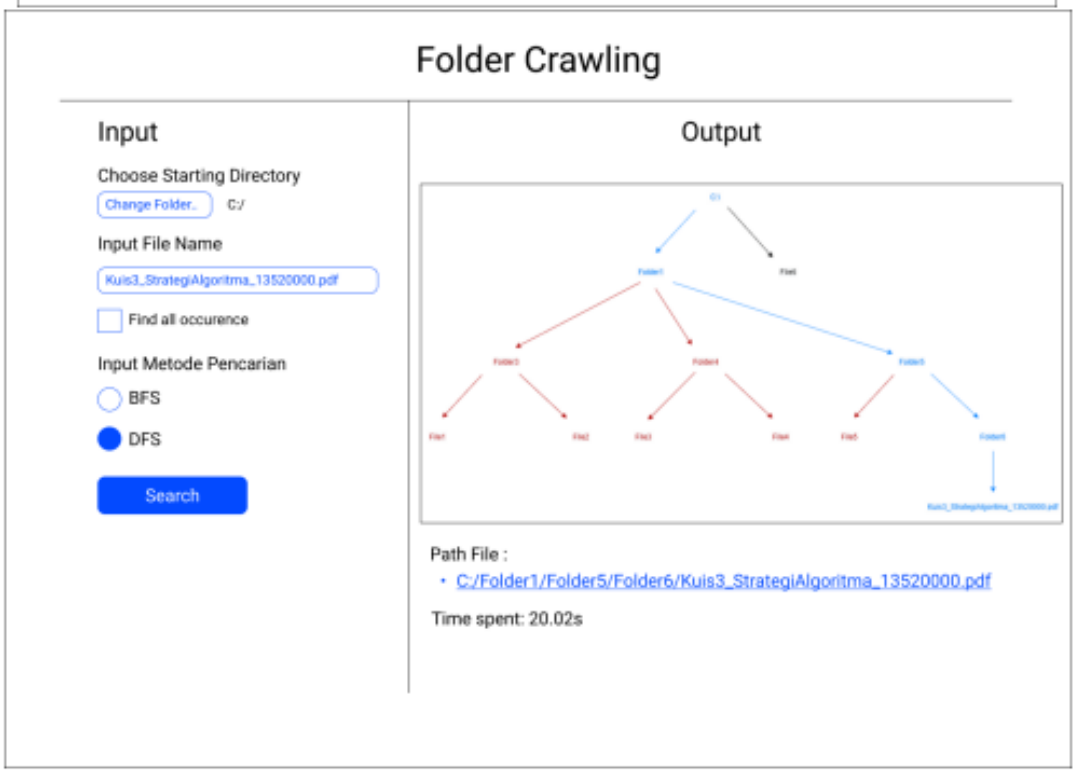


Gambar 5. Contoh Ketika hyperlink di-klik

Spesifikasi Program:

Aplikasi yang akan dibangun dibuat berbasis GUI. Berikut ini adalah contoh tampilan dari aplikasi GUI yang akan dibangun.





Gambar 6. Tampilan layout dari aplikasi desktop yang dibangun

BAB 2 – LANDASAN TEORI

A. *GRAPH TRAVERSAL* (BFS dan DFS)

Algoritma traversal graf adalah salah satu algoritma pencarian solusi dengan mengunjungi simpul dengan cara yang sistematik dengan asumsi graf saling terhubung. Terdapat dua representasi graf, yang pertama adalah graf statis yang sudah terbentuk sebelum dilakukannya proses pencarian dan kedua adalah graf dinamis yang terbentuk saat proses pencarian dilakukan. Graf statis sendiri memiliki dua pendekatan pencarian yaitu pencarian mendalam (DFS/*Depth First Search*) dan pencarian melebar (BFS/*Breadth First Search*).

Pencarian mendalam atau *depth first search* adalah proses pencarian pada graf secara mendalam. Pencarian dengan metode DFS mencari simpul hingga simpul terdalam dan melakukan *backtracking* ke simpul terakir yang dikunjungi sebelumnya jika tidak ada simpul yang bisa dicari lagi. Pencarian dengan metode DFS memiliki langkah-langkah sebagai berikut,

1. Kunjungi simpul *v.*
2. Kunjungi simpul *w* yang bertetangga dengan simpul *v*.
3. Ulangi DFS mulai dari simpul *w*.
4. Ketika mencapai simpul *u* sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian dirunut-balik (*backtrack*) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul *w* yang belum dikunjungi.
5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.

Pencarian melebar atau *breadth first search* adalah proses pencarian graf secara melebar. Berbeda dengan metode DFS, pencarian dengan metode BFS tidak perlu melakukan *backtracking* karena mengunjungi semua simpul tetangga terlebi dahulu sebelum mengunjungi simpul yang lebih dalam. Pencarian dengan metode BFS memiliki langkah-langkah sebagai berikut,

1. Kunjungi sebuah simpul *v.*
2. Kunjungi semua simpul yang bertetangga dengan simpul *v* terlebih dahulu.
3. Kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang tadi dikunjungi, demikian seterusnya.

B. C# *DESKTOP APPLICATION DEVELOPMENT*

Desktop application yang dibuat pada tugas ini akan menggunakan *Framework* .NET yang merupakan sebuah *software framework* yang dikembangkan oleh Microsoft yang utamanya berjalan pada sistem operasi Windows. Beberapa kelebihan dari *framework* ini adalah tidak bergantung pada *platformI* tertentu (dapat dijalankan pada *device* apapun yang telah ter-*install* implementasi dari .NET), dapat digunakan dengan banyak bahasa pemrograman, memiliki kecepatan eksekusi yang baik, dan yang terakhir *framework* ini bersifat *open source*.

Framework .NET terdiri dari CLR (*Common Language Runtime*) dan *Class Library*. CLR sebagai dasar dari *framework* ini, bertanggung jawab dalam mengelola kode pada proses ekseskusi berlangsung (pengalokasian memori, akurasi kode, dll). Sedangkan, *Class Library* adalah kumpulan berbagai *class*, *interface*, dan *value type* yang memungkinkan untuk melakukan berbagai tugas pemrograman umum. Bahasa C# merupakan salah satu bahasa pemrograman yang menggunakan *Class Library* dari *Framework* .NET ini. C# (C-*Sharp*) pada dasarnya merupakan hasil

pengembangan dari bahasa pemrograman C dan C++ sehingga sintaks dari bahasa ini masih memiliki banyak kemiripan dengan C dan C++.

Desktop application ini akan dibuat menggunakan GUI (*Graphical User Interface*) / antarmuka, yaitu suatu sistem yang menggunakan metode interaksi secara grafis antara pengguna dan komputer. Kerangka kerja GUI yang umumnya digunakan oleh *developer* .NET adalah WinForm dan WPF. GUI pada *desktop application* ini akan dibuat dengan menggunakan WPF (*Windows Presentation Foundation*). WPF merupakan versi yang lebih baru jika dibandingkan dengan WinForm sehingga bersifat lebih fleksibel dan memungkinkan penggunanya membuat *advance* UI (*User Interface*). WPF memungkinkan pemisahan pekerjaan antara *designer* (menggunakan XAML yang digunakan untuk merepresentasikan UI dan *user interaction*) dengan *programmer* (menggunakan bahasa C#, dll).

BAB 3 – ANALISIS PEMECAHAN MASALAH

A. LANGKAH-LANGKAH PEMECAHAN MASALAH

Pemecahan masalah pada aplikasi *Folder Crawler* dikerjakan dengan menggunakan dua metode algoritma pencarian yaitu BFS dan DFS

Metode pencarian dengan DFS:

a. Menerima input nama *file* yang ingin dicari beserta tempat awal *folder* yang akan dicari
b. Mencari *folder* atau *file* pertama yang terletak di *folder* awal lalu memasukkannya sebagai anak ke dalam pohon.
c. Membuka folder pertama tersebut dan mencari *folder* atau *file* pertama sama seperti sebelumnya lalu memasukkannya sebagai anak ke dalam pohon.
d. Jika sudah tidak ada *folder* yang bisa dibuka kemudian mengecek apakah *file* di dalam *folder* tersebut apakah sama dengan *file* yang ingin dicari. Tidak lupa memasukan semua *file* yang telah di cek sebagai anak ke dalam pohon.
e. Jika *file* yang dicek sama dengan *file* yang dicari maka pencarian dihentikan dan mengembalikan *path* dari *file* tersebut.
f. Jika tidak maka *folder* akan *backtracking* ke *folder parent* atau orang tua dan mencari di *folder* lainnya yang belum dicari
g. Langkah d dan f dilakukan terus menerus hingga *file* yang ingin dicari ditemukan atau tidak ada lagi *file* dan *folder* yang belum cek

Jika mode "*check for all occurence"* diaktifkan maka pada langkah e pencarian tidak akan dihentikan *path* dari *file* tersebut akan dimasukkan ke dalam *array* yang berisi *path* dari seluruh *file* yang sama.

Metode pencarian dengan BFS:

a. Menerima input nama *file* yang ingin dicari beserta tempat awal *folder* yang akan dicari
b. Membuat pohon direktori yang berisikan semua subentri dari *folder* awal tersebut.
c. Melakukan *enqueue* akar pohon direktori ke antrean pencarian.
d. Melakukan *dequeue* antrean dan memeriksa apakah dia bertipe direktori atau *file*.
e. Apabila bertipe *file*, *file* dibandingkan dengan nama *file* yang ingin dicari. Bila sama, pencarian dihentikan. Bila tidak, kembali ke langkah d.
f. Apabila bertipe direktori, subentri pada direktori tersebut di-*enqueue* ke antrean dan kembali ke langkah d.
g. Apabila antrean kosong, proses pencarian akan dihentikan.

Jika mode "*check for all occurence"* diaktifkan, pencarian tidak akan dihentikan pada langkah e apabila *file* ditemukan. *Path* dari *file* yang ditemukan tersebut akan dimasukkan ke dalam *array* yang berisi *path* dari seluruh *file* yang ditemukan.

B. MAPING PERSOALAN MENJADI ELEMEN ALGORITMA BFS DAN DFS

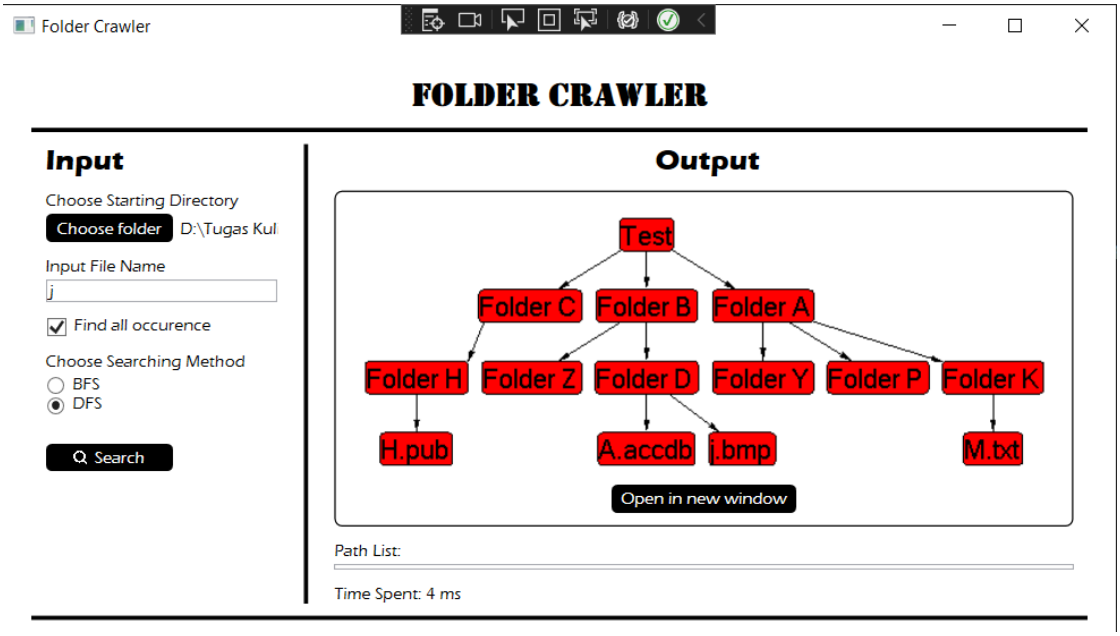
Persoalan *folder-crawler* menggunakan graf terutama graf berbentuk pohon untuk melambangkan prosesnya. Simpul pada graf melambangkan *folder* atau *file* yang minimal memiliki satu *parent* folder. Relasi hubungan antara *folder* atau *file* dengan *folder parent* atau *child* dilambangkan sebagai busur. Kemudian, terdapat sebuah pohon yang memiliki *key node*. Setiap

key node memiliki tipe data yaitu *data* yang berisi nama *file* atau *folder*, *colour* bertipe *integer* yang menunjukkan apakah simpul tersebut mengarah ke solusi atau tidak, *path* yang berisi *Full Path* dari file tersebut dan *children* yang berisi simpul-simpul *file* atau *folder* yang berada di dalam *folder key node* tersebut. Himpunan kandidat untuk semua persoalan adalah semua *file* atau *folder* yang berada pada *folder* yang dicari. Secara khusus *mapping* persoalan untuk fitur pada *folder crawler* adalah sebagai berikut,

a. *Folder Crawler* dengan DFS
   Berdasarkan himpunan kandidat yang ada dilakukan pencarian *file* dari *folder* terpilih. Jika *file* yang terpilih tidak ditemukan maka akan mencari *file* berikut di *folder* pertama dalam *folder* terpilih. Pencarian dengan DFS ini memanfaatkan struktur data *tree* yang akan menyimpan simpul yang telah dikunjungi. Jika *"found all occurence"* diaktifkan maka setiap *file* yang ditemukan akan dimasukkan ke dalam *array* yang berisi seluruh *path file* hasil pencarian. Himpunan solusi dari fitur *folder crawler* adalah *tree* yang berisi simpul-simpul yang telah dikunjungi.
b. Folder Crawler dengan BFS
   Pencarian BFS memanfaatkan struktur data *queue* yang menyimpan entri dari pohon direktori. Berdasarkan himpunan kandidat yang ada, *file* dan *folder* di dalam *folder* terpilih berturut-turut akan dimasukkan ke dalam antrean. Isi antrean akan di-*dequeue* dan diperiksa jenisnya. Bila isi antrian tersebut berjenis *file*, pencarian akan dihentikan bila ditemukan kecocokan dengan *file* yang dicari. Apabila tidak ditemukan kecocokan, isi antrean akan di-*dequeue* kembali dan proses berulang. Apabila isi antrean tersebut berjenis *file*, *file* dan *folder* di dalam *file* tersebut akan dimasukkan ke dalam antrean. Proses ini berlangsung hingga antrean kosong. Jika "*found all occurence*" diaktifkan, proses pencarian tidak akan berhenti ketika *file* ditemukan dan penyimpanan *path file* yang ditemukan akan memanfaatkan *array*. Himpunan solusi dari *folder crawler* adalah pohon direktori dengan simpul berupa entri-entri yang telah dikunjungi atau dijadwalkan untuk dikunjungi.

C. ILUSTRASI KASUS

a. *File* yang dicari tidak ditemukan
   Kasus lain adalah tidak ditemukannya *file* yang sedang dicari. Contoh kasus tersebut dapat diilustrasikan sebagai berikut,



Gambar 7. Tampilan aplikasi untuk kasus file yang dicari tidak ditemukan

Ketika mencari *file* bernama "j", Di dalam *folder test* berserta turunannya tidak ditemukan *file* dengan nama tersebut. Sehingga pohon yang dihasilkan semua berwarna merah dan *Path List* berisi kosong.

BAB 4 – IMPLEMENTASI DAN PENGUJIAN

A. IMPLEMENTASI PROGRAM

Karena menggunakan WPF, tidak ada program utama yang dituliskan secara eksplisit melainkan hanya terdapat implementasi fungsi-fungsi yang akan dieksekusi jika terdapat *event* tertentu (umumnya saat menekan tombol). Hal ini diimplementasikan pada *file* `MainWindow.xaml.cs` dan `MainWindow.xaml` di *folder* GUI. Jadi, seperti yang sudah dijelaskan di dasar teori, *file* dengan ekstensi `.xaml` akan berisi desain UI dari aplikasi sehingga isinya adalah kumpulan *tag* berisi objek *button*, *textbox*, *image*, *radio button*, *checkbox*, dan lain-lain. Pada setiap deklarasi *tag button*, akan disisipkan pemanggilan ke suatu fungsi yang bersesuaian pada *file* `MainWindows.xaml.cs`, misalnya *button* "Choose folder" akan mengeksekusi fungsi `BtnChooseFolder_Click()` yang akan menampilkan sebuah jendela *file explorer* untuk menentukan *starting directory* pencarian yang diinginkan *user*.

Berikut ini adalah *pseudocode* fungsi untuk masing-masing tombol (*button*) yang ada:

a. Tombol *Search* (btnSearch)

Fungsi ini merupakan fungsi utama yang akan mengumpulkan semua *input* dari *user*, melakukan pemanggilan fungsi DFS dan BFS, serta menampilkan *output* hasil pencarian ke layar aplikasi.

```
procedure BtnSearch_Click(object sender, RoutedEventArgs e)
{ Mengeksekusi program ketika menekan search }
    bool inputValid ← cek apakah semua form input sudah terisi dan
valid
    { Jika Valid }
    if (inputValid) then
        { Hapus semua output bekas eksekusi sebelumnya }
        ClearOutputScreen()

        { Variabel Input }
        string start ← nama directory yang ada pada textbox
chooseFolder
        DirectoryInfo diSource ← convert start jadi tipe DirectoryInfo
        string fileName ← nama file yang ada pada textbox inputFileName
        bool Occurence ← cek apakah checkbox all accourence dipilih

        { Variabel Output }
        NTree<FileSystemInfo> pohon
        List<string> path
        long elapsedTime

        { ALGORITMA BFS dan DFS }
        if (tombol BFS dipilih) then
            BreadthFirstSearch bfsSearch ← membuat objek Breatdh Fisrst
Search dengan parameter occurrence
            { Panggil fungsi DFS }
            (path, pohon, elapsedTime) ← Hasil Algoritma DFS

        else if (tombol DFS dipilih) then
            Stopwatch stopWatch
            Mulai stopwatch
            found found ← membuat objek found dengan parameter bernilai
false
            pohon ← Hasil algotitma DFS
            Hentikan stopwatch
```

```
            elapsedTime ← waktu eksekusi

        { Output }
        { Menampilkan gambar pohon }
        Gambar tree dengan fungsi ViewerSample.drawTree(pohon)
        Tampilkan gambar bitmap ke layar aplikasi

        { Menampilkan path dari file yang dicari }
        if (jumlah path yang ditemukan > 0) then
            Tampilkan keterangan "(Double click to open folder)"
        else if (jumlah path yang ditemukan = 0) then
            Tampilkan keterangan "FILE NOT FOUND!"

        i traversal [0..jumlah path yang ditemukan-1]
            Tampilkan path[i] ke layar aplikasi

        { Menampilkan waktu yang diperlukan selama pencarian }
        Tampilkan nilai variable elapsedTime ke layar aplikasi

        { Menampilkan tombol untuk membuka window baru untuk gambar }
        Tampilkan btnOpenInNewWindow
```

b. Tombol *Choose Folder* (btnChooseFolder)
Fungsi ini digunakan untuk menampilkan *file explorer* dan memilih *starting directory* untuk memulai pencarian.

```
procedure BtnChooseFolder_Click(object sender, RoutedEventArgs e)
{ Menampilkan window file explorer ketika menekan tombol search }
    var openFolderDialog { Membuat objek OpenFolderDialog }
    Tampilkan jendela file explorer
    Isi textbox chooseFolder dengan starting directory yang dipilih
```

c. Tombol *Open in New Window* (btnOpenInNewWindow)
Fungsi ini digunakan untuk menampilkan gambar pohon di jendela baru (WinForm). Tombol ini hanya akan muncul pada tampilan layar setelah pencarian selesai dilakukan.

```
procedure BtnOpenInNewWindow_Click(object sender, RoutedEventArgs e)
{ Membuka window baru untuk menampilkan tree }
    { create a form }
    Form form
    { associate the viewer with the form }
    Hentikan semua proses pada form untuk sementara waktu
    Masukkan gambar pohon ke form
    Lanjutkan proses pada form
    { show the form }
    Tampilkan jendela baru / WinForm
```

d. *Hyperlink path file*
Fungsi ini digunakan untuk meng-*hyperlink* ke *folder* tempat *file* yang dicari tersimpan.

```
procedure PathFile_Click(object sender, RoutedEventArgs e)
{ Hyperlink ke path file yang diklik }
    if (isi path yang di-double click ≠ null) then
        string filePath ← convert path file menjadi string
        string folderPath ← ambil directory folder dari filepath
        if (folderPath valid) then
            Buka jendela file explorer pada directory folderPath
```

Algoritma DFS dan BFS diimplementasikan masing-masing pada *file* DFS.cd di *folder* DFS dan *file* BFS.cs di *folder* BFS. Pencarian BFS dan DFS memiliki algoritma yang serupa. Perbedaanya adalah BFS menggunakan *queue* untuk menyimpan urutan pengecekan *node-node*, sedangkan DFS hanya menggunakan *tree.*

Berikut algoritma dari BFS dengan *queue,*

```
function SearchFile(string targetFile)
    enqueue _searchTree to _searchQueue
    levelCountdown <- 1
    directoryCounter <- 0

    while _searchQueue is not empty and (file is not found or look for
                                         multiple occurence))
        dequeue _searchQueue to currentEntry
        depend on currentEntry.data
            case FileInfo file when file.Name equals targetFile:
                _fileFound <- true;
                add file.FullName to _filePaths
            case DirectoryInfo:
                for each entry in currentEntry.children
                    enqueue entry to _searchQueue
                    if entry.data is DirectoryInfo
                        increment directoryCounter

                decrement levelCountdown
                if levelCountdown equals 0
                    increment _fileFoundDepth
                    levelCountdown <- levelCountdown + directoryCounter
                    directoryCounter <- 0
```

Berikut algoritma dari DFS dengan *tree,*

```
function searchFolder(DirectoryInfo source, string target, List<string>
path, found found, bool occurence)
{ Membuka class Ntree baru dengan atribut source dan angka 2 }
    if (Hanya cari occurence pertama dan file sudah ditemukan) then
        → tree
    for each (file ∈ source.GetDirectories())
        NTree<FileSystemInfo> file ← searchFolder(diSourceSubDir, target,
path, found, occurence)
        Masukkan file sebagai children dari tree
        if (Cari semua occurence) then
            if (nilai colour dari file adalah 1) then
                tree.colour ← 1
            else if (nilai colour dari tree adalah 2 dan file adalah 0)
then
                tree.colour ← 0
        else if (Hanya cari occurence pertama) then
            if (file belum ditemukan) then
                tree.colour ← 0
            else if (file sudah ditemukan dan colour dari file adalah 1)
then
                tree.colour ← 1
            else if (file sudah ditemukan dan colour dari file adalah 0
tapi dari tree bukan 1) then
                tree.colour ← 0
```

```
    for each (fi ∈ source.GetFiles())
        if (cari semua occurence) then
            if (file sama dengan fi) then
                Masukkan path dari fi kedalam linked list path
                Masukkan fi sebagai children dari tree dengan colour 1
                tree.colour ← 1
                Ubah nilai found.find menjadi true
            else
                Masukkan fi sebagai children dari tree dengan colour 0
                if (colour dari tree bukan 1) then
                    tree.colour ← 0
        else if (Hanya cari occurence pertama) then
            if (file sama dengan fi dan file belum pernah ketemu
sebelumnya) then
                Masukkan path dari fi kedalam linked list path
                Masukkan fi sebagai children dari tree dengan colour 1
                tree.colour ← 1
                Ubah nilai found.find menjadi true
            else if (file sudah ketemu) then
                Masukkan fi sebagai children dari tree dengan colour 2
            else if (file belum ketemu) then
                Masukkan fi sebagai children dari tree dengan colour 0
                tree.colour ← 0
        if (nilai colour dari tree adalah 2) then
            tree.colour □0
        → tree
```

B. STRUKTUR DATA PROGRAM

BFS dan DFS menggunakan *class* NTree dalam melakukan algoritma pencarian *file*. *Class* NTree adalah *class* dengan struktur data berdasar dari *Tree*. *Class* NTree memiliki 3 buah atribut yaitu `data` (untuk menampung informasi data bertipe T pada *node* tersebut), `colour` (untuk menampung informasi warna *node* tersebut, yaitu 0 untuk merah, 1 untuk biru, dan 2 untuk hitam), dan `children` (untuk menampung anak/*child* yang juga bertipe NTree, dari *node* saat ini). Berikut adalah *method* dari *class* NTree,

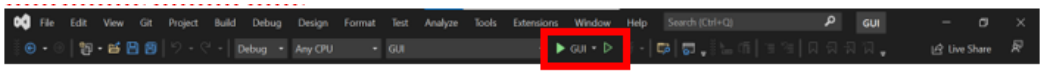
```
constructor NTree(T data, int colour)
    this.data ← data
    this.colour ← colour
    Children ← new LinkedList<NTree<T>>()

procedure AddChild(T data, int colour)
    children.AddFirst(new NTree<T>(data,colour))

procedure Traverse(NTree<T> node, int i)
    print (node.colour +". " +node.data)
    for each (kid ∈ node.children)
        Traverse(kid, i+1)
```

C. TATA CARA PENGGUNAAN PROGRAM

Untuk menjalankan aplikasi *folder crawler*, silakan buka (*run*) *file* `FolderCrawler.exe` yang berada di *folder* `bin`. Atau apabila ingin melakukan *build* ulang, bisa dilakukan menggunakan Visual Studio dengan langkah langkah sebagai berikut:

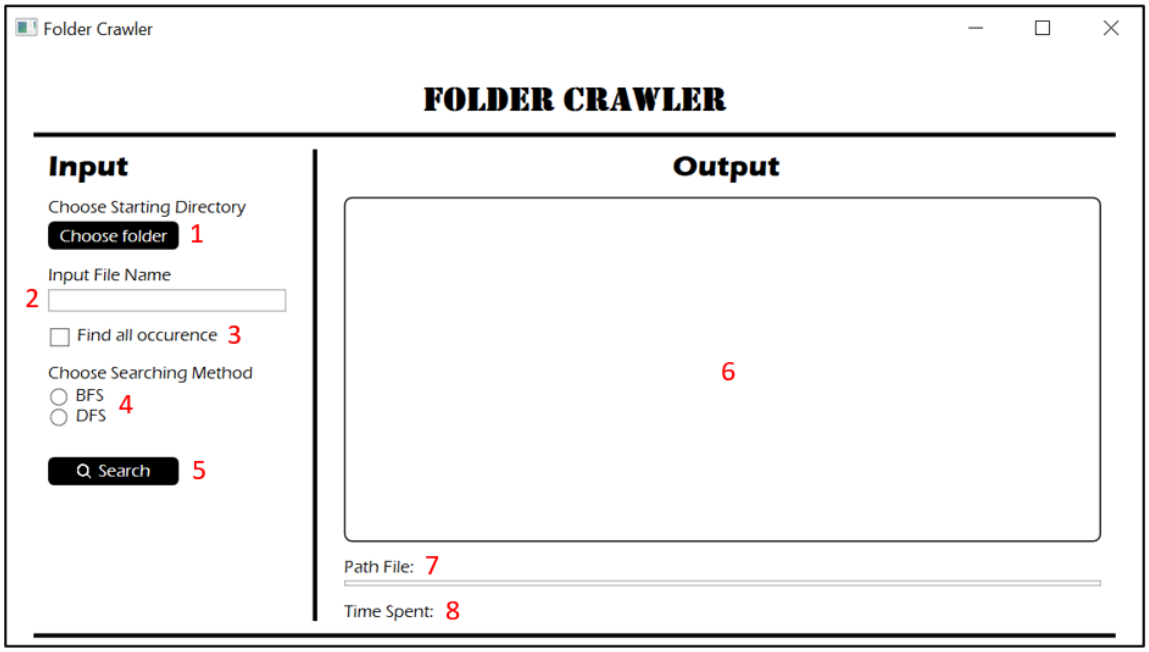
1. Buka *file* `GUI.sln` yang berada di *relative path* `scr/GUI/` menggunakan Visual Studio

2. Lakukan *build* terhadap GUI dengan memilih menu Build >> Build GUI
3. Untuk menjalankan program, terdapat 2 pilihan, yaitu:
   a. Buka *directory* `src/GUI/bin/Debug/net6.0-windows/` lalu jalankan *file* `GUI.exe`
   b. Pilih tombol segitiga hijau (*Run*) di bagian atas (menu) seperti pada gambar di bawah ini



4. Setelah itu, akan muncul jendela aplikasi *folder crawler* dan program siap dijalankan.

Fitur yang disediakan oleh aplikasi ini adalah fitur pencarian dengan 2 metode yaitu *Depth First Search* dan *Breadth First Search* serta fitur menemukan semua kemunculan *file* atau menemukan kemunculan *file* pertama. Secara umum, *interface*/tampilan layar aplikasi dibagi menjadi 2 bagian yaitu panel *input* (kiri) dan panel *output* (kanan). Gambar di bawah ini merupakan tampilan GUI untuk aplikasi *folder crawler* yang dibuat.



Gambar 8 Tampilan GUI aplikasi folder crawler

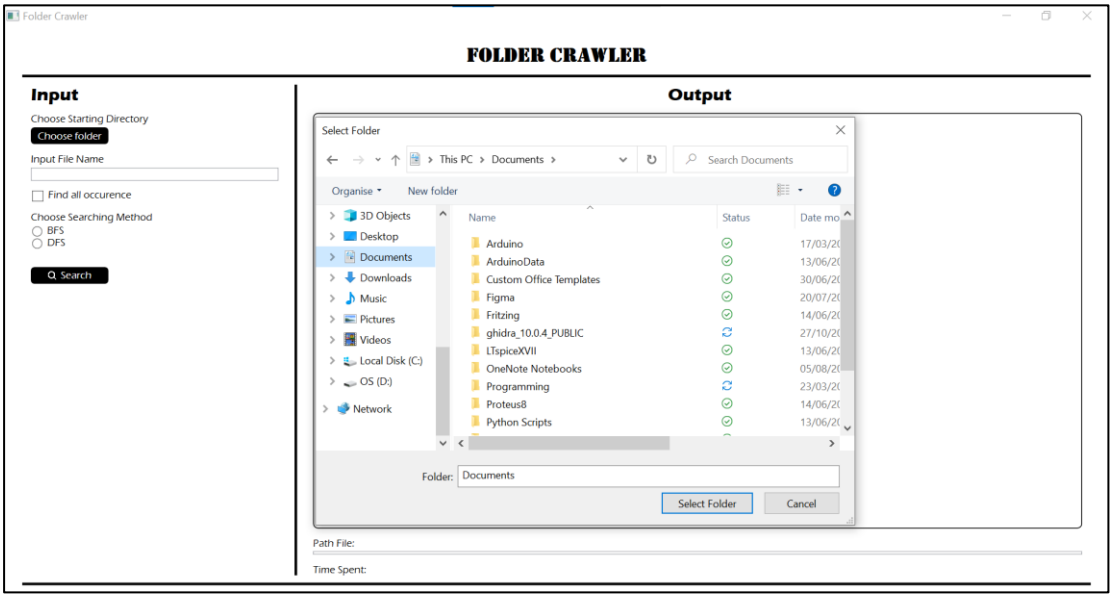
Deskripsi mengenai objek-objek yang terdapat pada tampilan diatas adalah sebagai berikut:

1. Tombol *Choose Folder* digunakan untuk memunculkan jendela *file explorer* yang digunakan untuk memilih *starting directory* pada pencarian *file*.
2. *TextBox* digunakan untuk menerima input nama *file* yang akan dicari.
3. *CheckBox Find All Occurrence* digunakan untuk menentukan apakah ingin mencari semua kemunculan *file* atau hanya kemunculan *file* pertama.
4. *Radio Button* BFS dan DFS digunakan untuk menentukan metode *folder crawling* yang akan digunakan.
5. Tombol *Search* digunakan untuk memulai pencarian *file* berdasarkan *input* yang diterima di atas.
6. *Rectangle* / kotak digunakan untuk menampung *output* gambar *tree* hasil *folder crawling*.
7. *ListBox Path File* digunakan untuk menampung seluruh *path* ke *file* yang dicari jika ditemukan.
8. *TextBlock Time Spent* digunakan untuk menampung *output* lamanya waktu yang diperlukan untuk melakukan pencarian *file*.

9. Tombol *Open in New Window* digunakan untuk menampilkan gambar *tree* di jendela baru (dalam bentuk WinForm). Tombol ini tidak terlihat pada tampilan awal, namun akan terlihat setelah gambar *tree* dimunculkan.

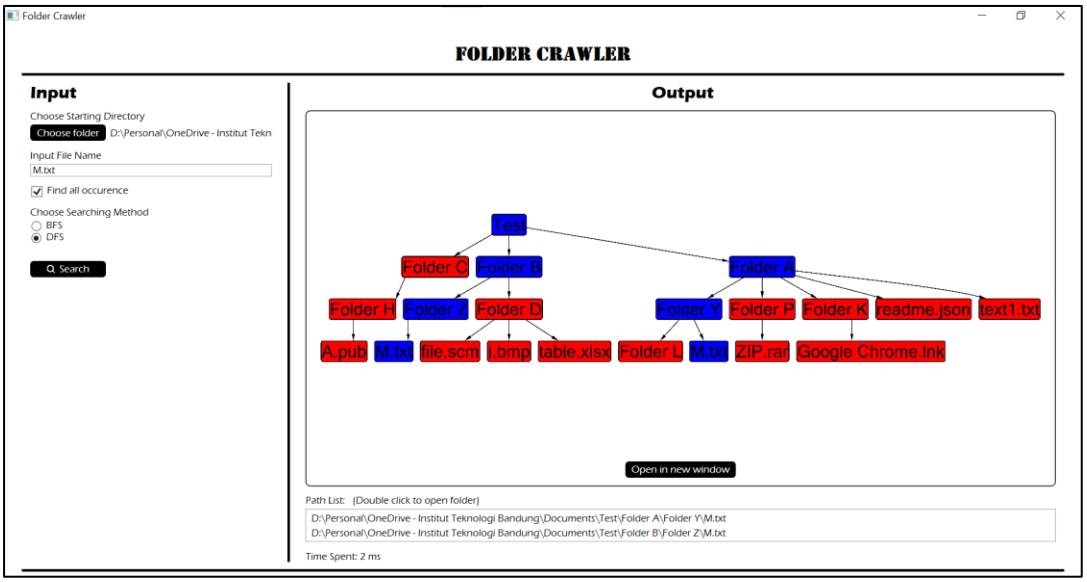
Tata cara penggunaan aplikasi *folder crawler* ini adalah sebagai berikut:

1. Masukkan *input starting directory* dengan menekan tombol *choose folder*. Setelah ditekan, akan muncul jendela *file explorer*, selanjutnya silakan pilih *folder* yang ingin dijadikan *starting directory*.



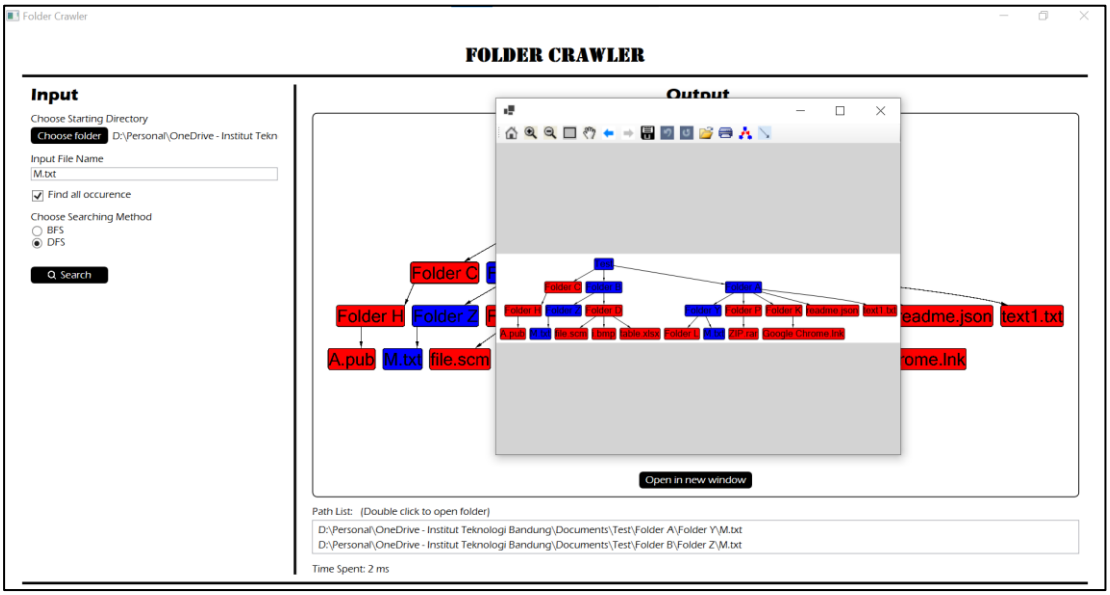
Gambar 9 Tampilan ketika ingin memasukkan starting directory

2. Masukkan nama *file* yang dicari, mode pencarian (semua kemunculan atau tidak), dan metode pencarian (BFS atau DFS).
3. Tekan tombol *Search* untuk memulai pencarian
4. Hasil pencarian akan dimunculkan pada panel *output* (bagian kanan) yang terdiri atas gambar pohon pencarian, *list path file*, dan waktu eksekusi. Gambar *tree* akan otomatis tersimpan di *folder bin* setiap kali pencarian dilakukan (tombol *search* ditekan)



Gambar 10 Tampilan input dan output

5. Apabila gambar yang ditampilkan terlalu kecil atau kurang jelas, pengguna dapat menekan tombol *Open in New Window* untuk membuka pohon pencarian pada suatu jendela baru (WinForm). Pada jendela WinForm ini, pengguna dapat memperbesar dan menggeser-geser gambar, menyimpan gambar, dll.

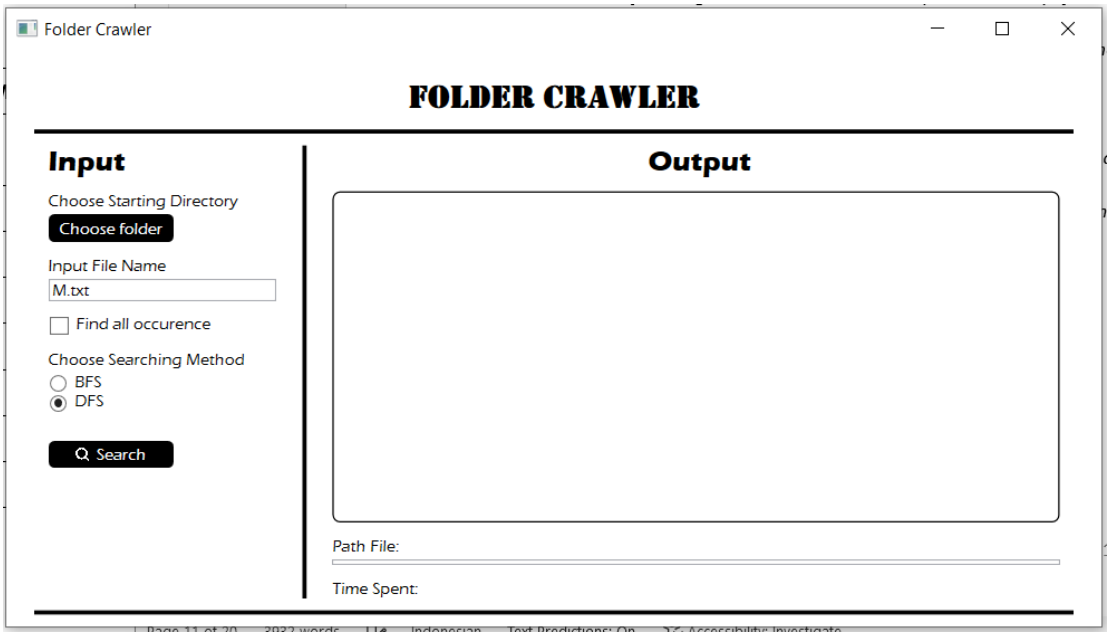


Gambar 11 Tampilan jendela WinForm setelah menekan tombol open in new window

D. HASIL PENGUJIAN

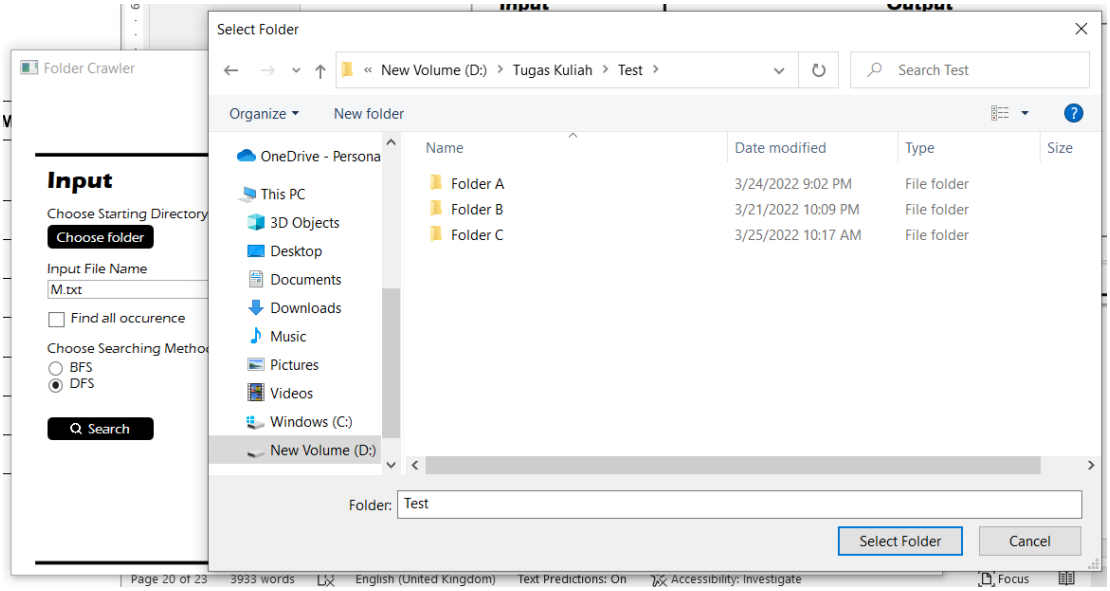
Berikut ini adalah beberapa alternatif skenario pengujian yang kami lakukan:

1. Menekan tombol *Search* tetapi belum mengisi semua *field input* yang diminta



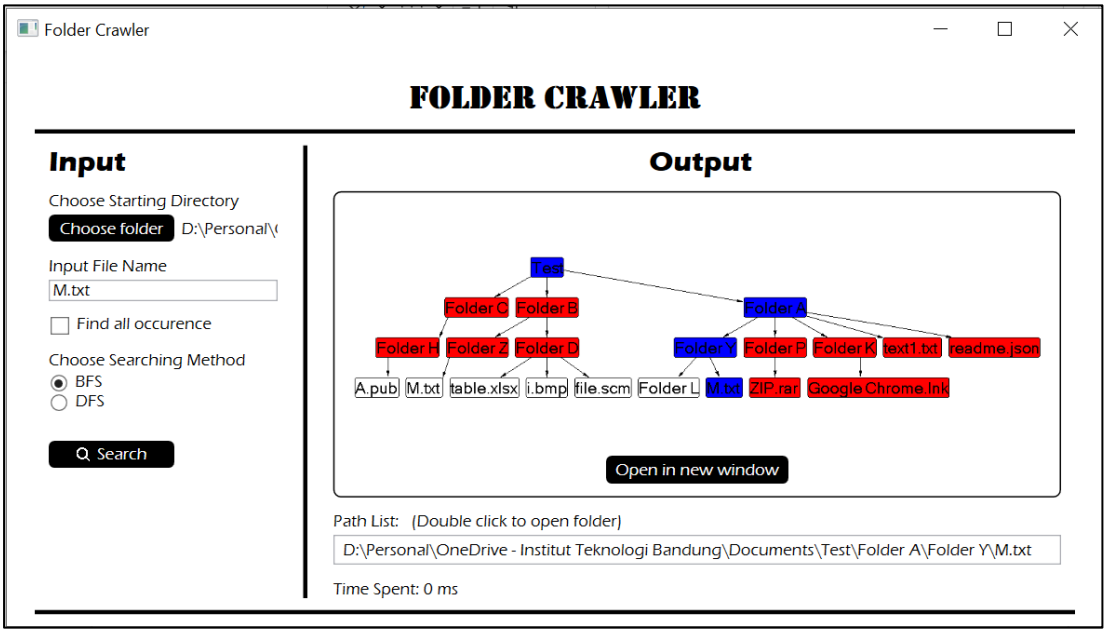
Gambar 12 Skenario 1

2. Menekan tombol *Choose Folder*

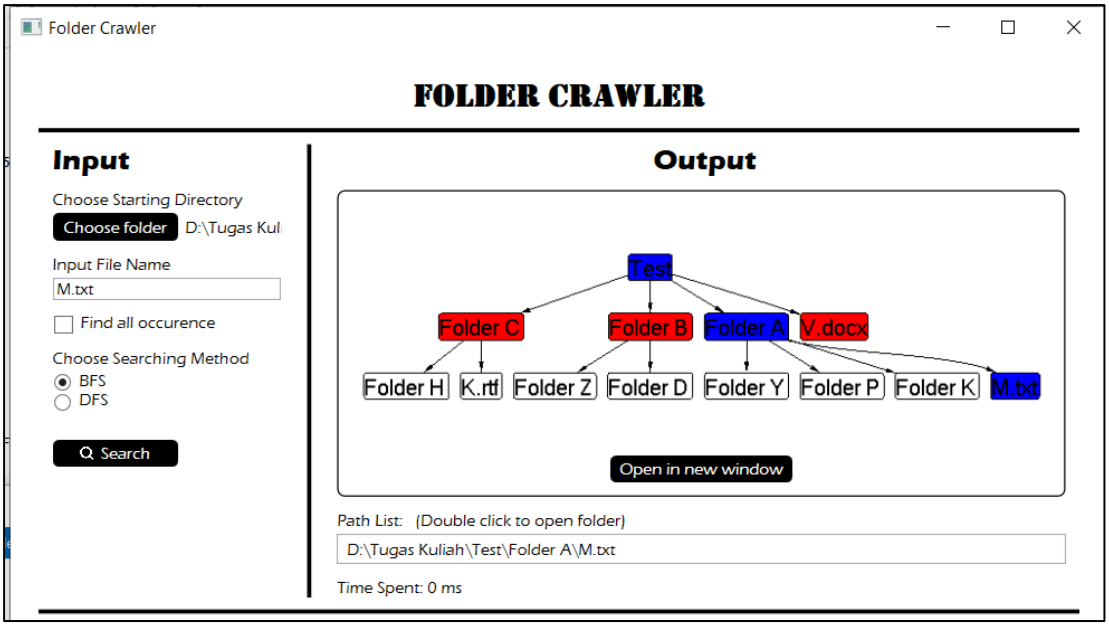


Gambar 13 Skenario 2

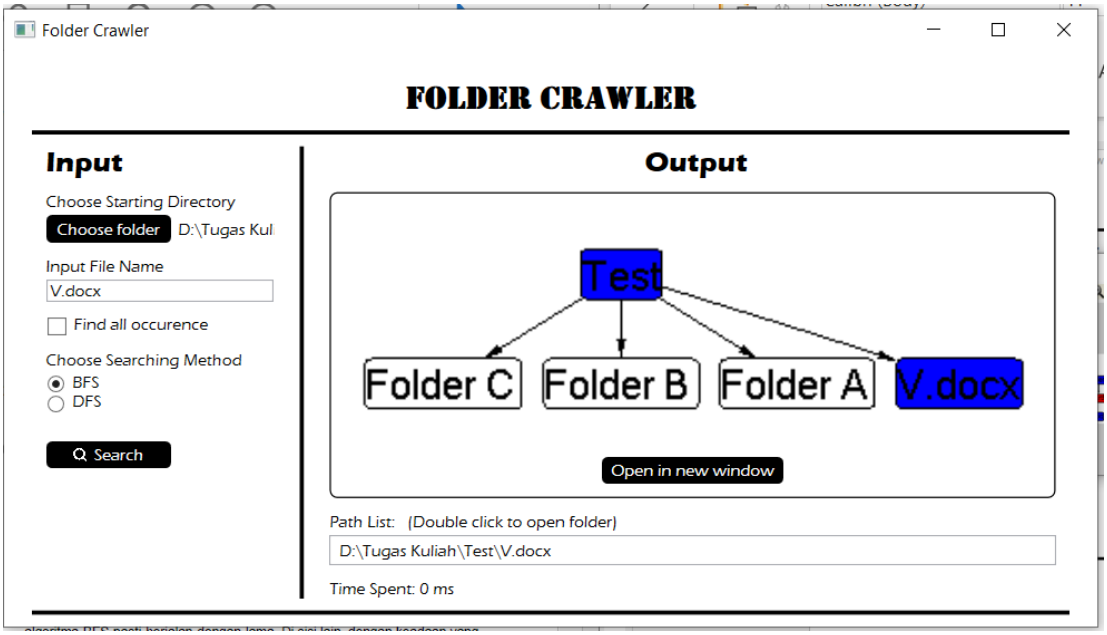
3. Pencarian *file* dengan metode BFS dan mode pencarian hanya *first occurrence*



Gambar 14 Skenario 3.1

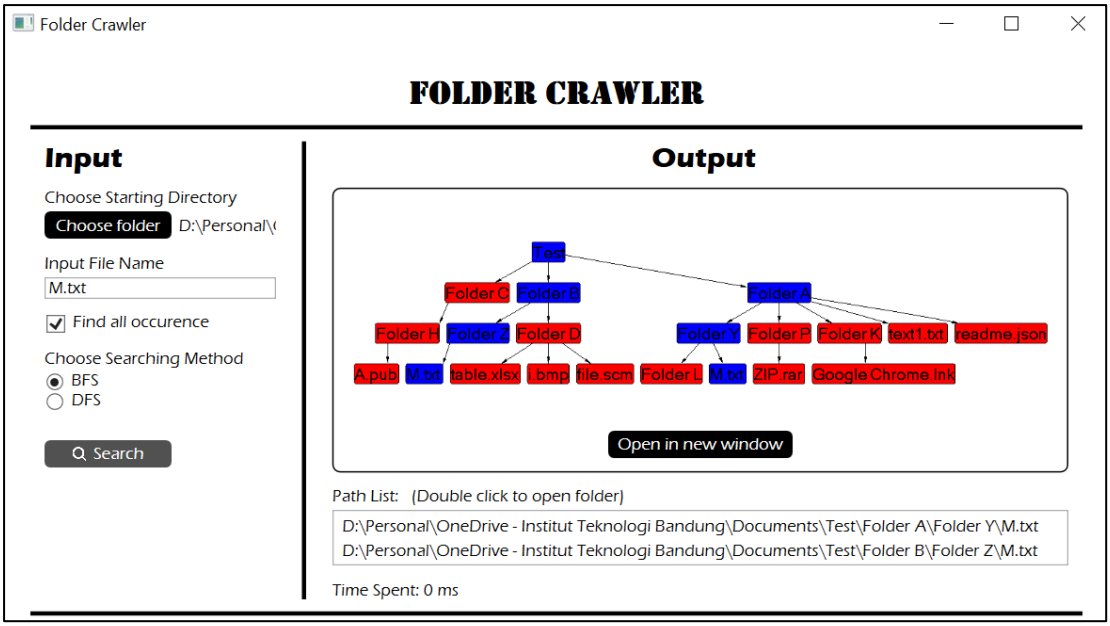


Gambar 15 Skenario 3.2



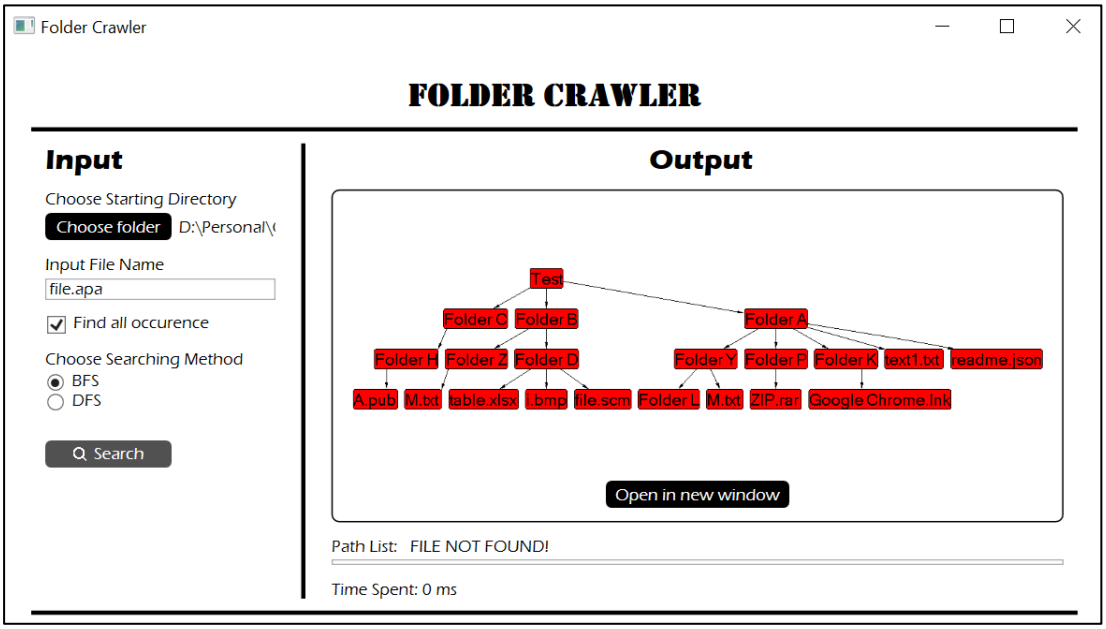
Gambar 16 Skenario 3.3

4. Pencarian *file* dengan metode BFS dan mode pencarian *all occurrence*



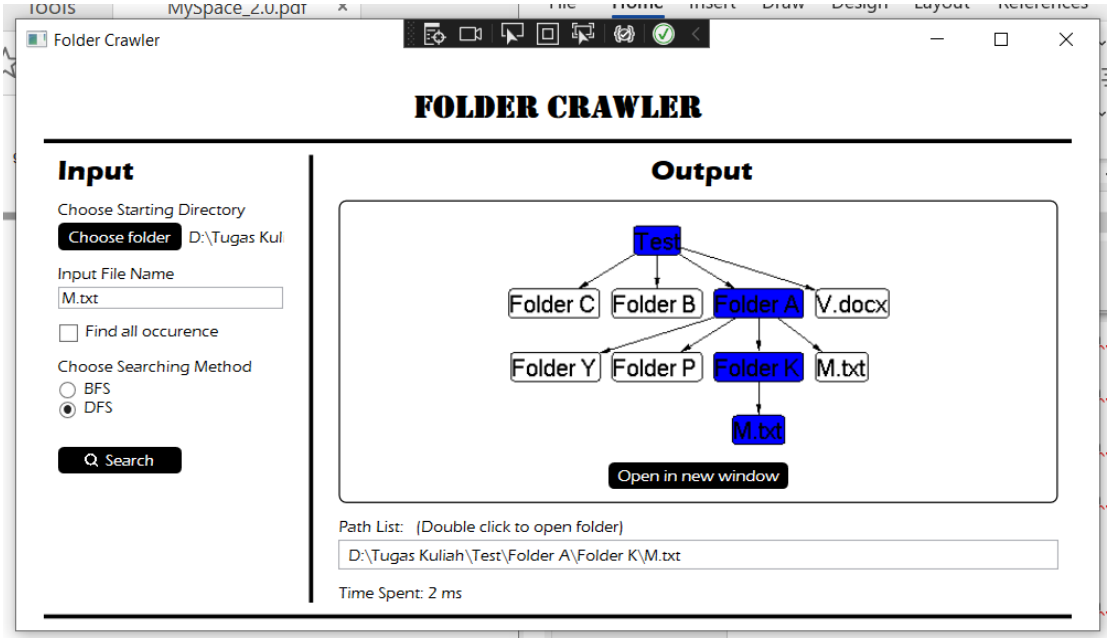
Gambar 17 Skenario 4

5. Pencarian *file* dengan metode BFS tetapi *file* tidak ditemukan

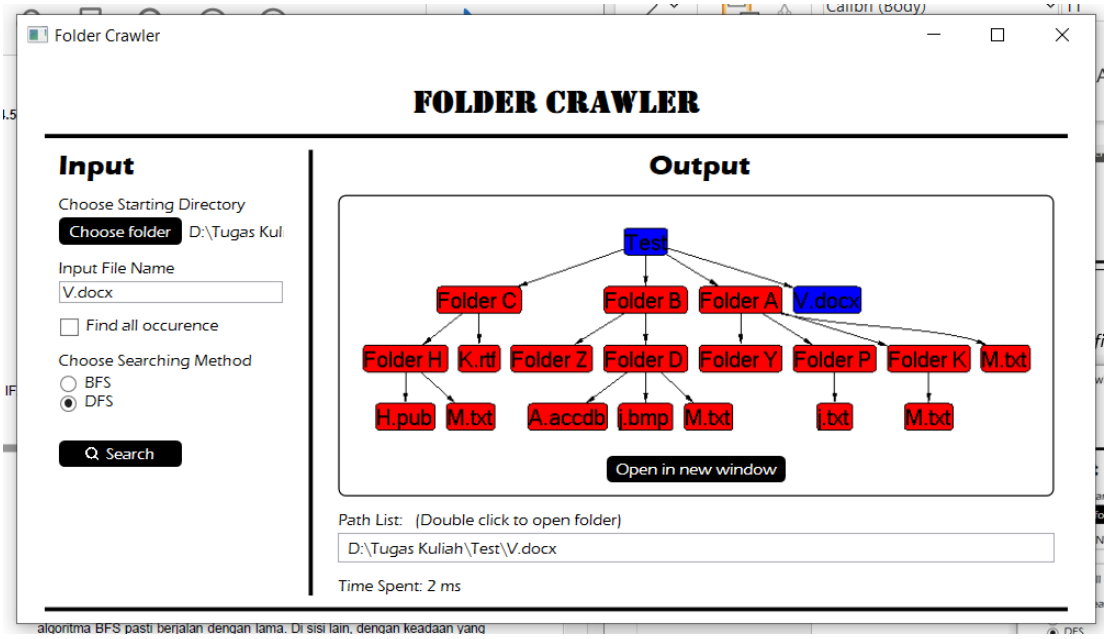


Gambar 18 Skenario 5

6. Pencarian *file* dengan metode DFS dan mode pencarian hanya *first occurrence*

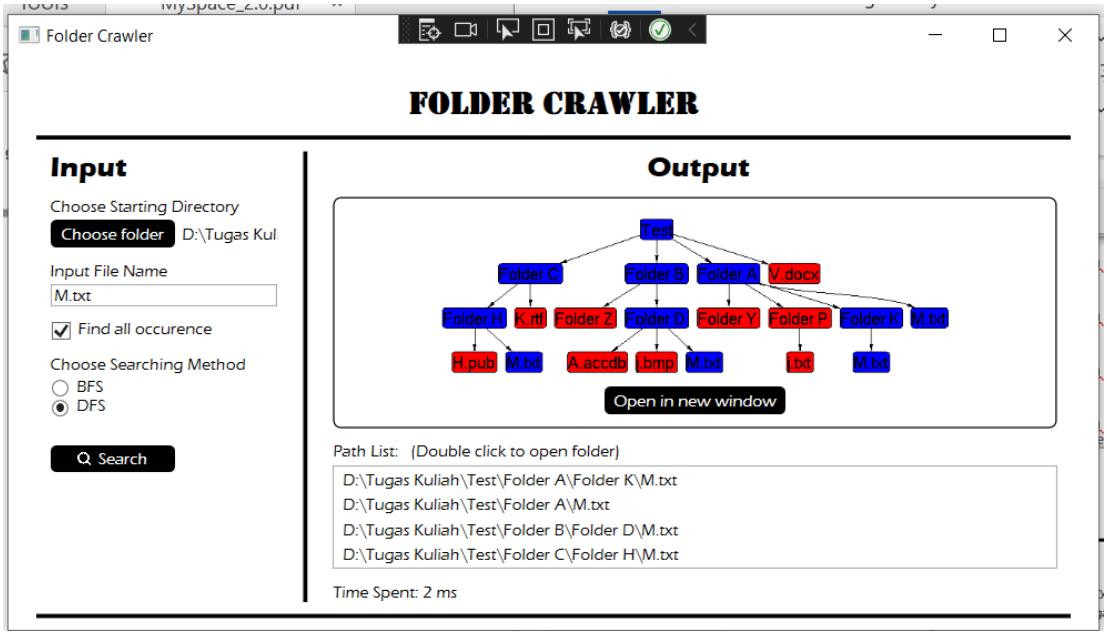


Gambar 19 Skenario 6.1



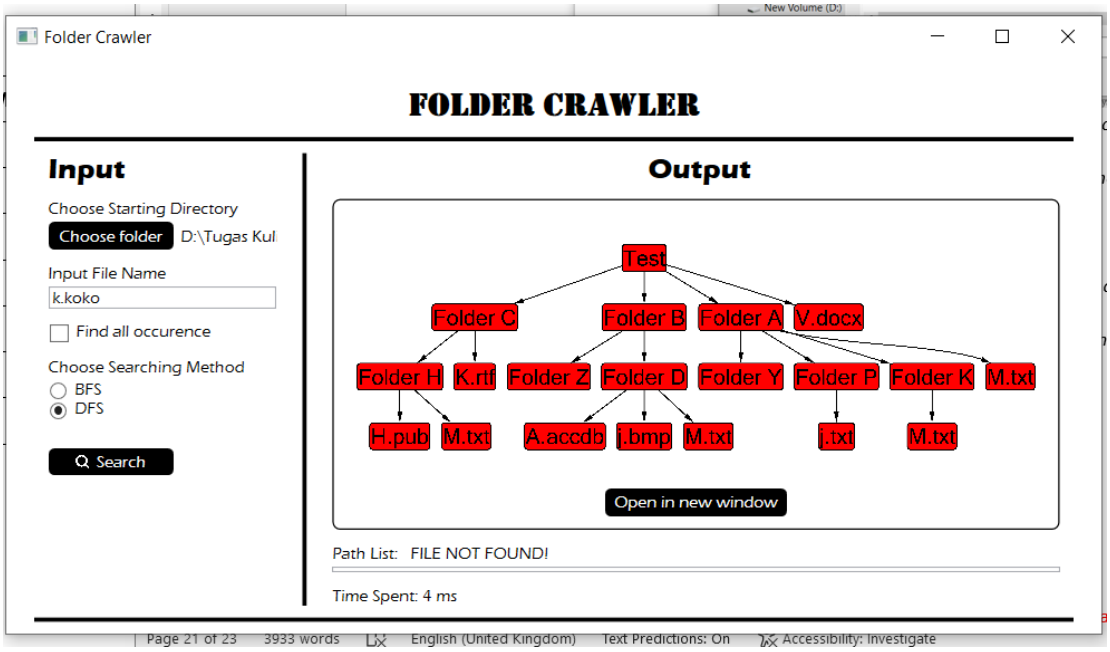
Gambar 20 Skenario 6.2

7. Pencarian *file* dengan metode DFS dan mode pencarian *all occurrence*



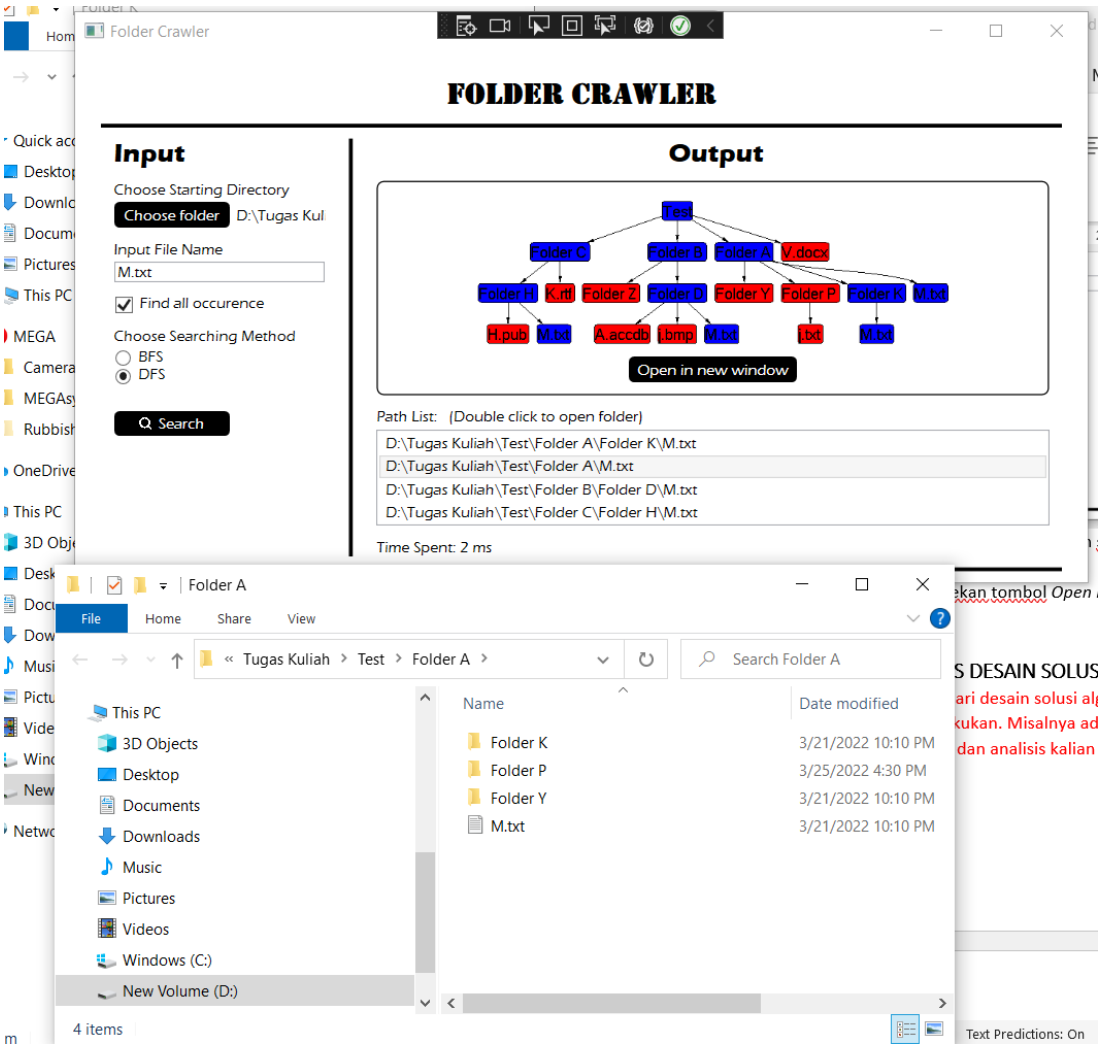
Gambar 21 Skenario 7

8. Pencarian *file* dengan metode DFS tetapi *file* tidak ditemukan



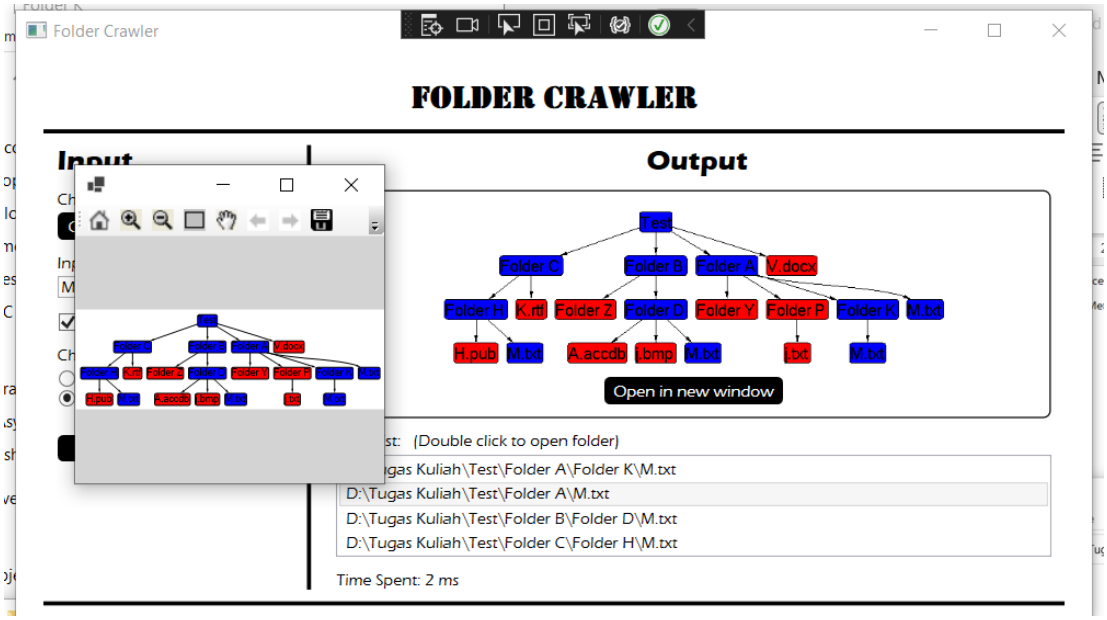
Gambar 22 Skenario 8

9. *Double click* pada salah satu *path file* yang ditemukan



Gambar 23 Skenario 9

10. Menekan tombol *Open in New Window*



Gambar 24 Skenario 10

E. ANALISIS DESAIN SOLUSI

Analisis pertama dilakukan dengan membandingkan pengujian pencarian *file* dengan metode *first occurence* pada pengujian ketiga dan keenam bagian D. Pada gambar 15, BFS membutuhkan 6 kali pengecekan *folder* dan *file* untuk menemukan *file* yang dicari. Sedangkan pada gambar 18, DFS hanya membutuhkan tiga kali pengecekan untuk menemukan *file* yang dicari. Terlihat bahwa pencarian dengan metode DFS lebih cepat daripada dengan metode BFS jika *file* yang dicari berada cukup dalam di dalam *folder-folder* lainnya. Hal itu karena DFS fokus untuk mencari ke dalam *folder* terlebih dahulu daripada mengecek semua isi dari *folder*.

Analisis kedua dilakukan dengan membanding pengujian pencarian *file* dengan metode *first occurence* pada pengujian ketiga dan keenam bagian D. Pada gambar 16, BFS hanya membutuhkan dua kali pengecekan sebelum menemukan *file* yang dicari. Sedangkan pada gambar 19, DFS membutuhkan 20 kali pengecekan sebelum menemukan *file* yang dicari. Terlihat bahwa pencarian dengan metode BFS lebih cepat daripada dengan metode DFS jika *file* yang dicari berada tidak terlalu dalam di folder-folder lainnya. Hal itu karena BFS fokus untuk mengecek semua isi dari *folder* terlebih dahulu baru mencari ke *folder* yang lebih dalam.

Analisis ketiga dilakukan dengan membandingkan pengujian pencarian *file* dengan metode *all occurence* pada pengujian keempat dan ketujuh bagian D. Berdasarkan gambar 17 dan 21 terlihat bahwa BFS dan DFS perlu mengecek semua isi *folder* sehingga kedua metode memiliki jumlah pengecekan yang sama. Oleh karena, itu dalam metode *all occurence* tidak kecepatan pencarian tidak dipengaruhi oleh BFS dan DFS.

Berdasarkan penjelasan di atas, dapat disimpulkan bahwa algoritma BFS lebih baik apabila *file* yang dicari terletak di kedalaman rendah. Sebaliknya, algoritma DFS lebih baik apabila *file* yang dicari terletak di kedalaman tinggi. Keunggulan algoritma BFS adalah nilai pencarian yang dihasilkannya konsisten, sedangkan keunggulan algoritma DFS adalah mungkin berjalan cepat walaupun pada *folder* besar.

BAB 5 – KESIMPULAN DAN SARAN

A. KESIMPULAN

Algoritma traversal graf merupakan algoritma pencarian solusi yang bekerja dengan cara mengunjungi simpul-simpul pada graf terhubung secara sistematis. Pada graf statis, algoritma traversal yang umum digunakan adalah algoritma *breadth-first search* (BFS) dan algoritma *depth-first search* (DFS). Algoritma BFS memanfaatkan struktur data *queue* untuk memeriksa simpul-simpul tetangga pada sebuah simpul sebelum berlanjut ke kedalaman selanjutnya. Algoritma DFS memanfaatkan struktur data *stack* untuk memeriksa upagraf terdalam dari sebuah simpul dan melakukan proses *backtracking*.

Aplikasi *folder crawler* yang telah kami implementasikan dapat melakukan traversal pohon direktori dengan baik. Implementasi algoritma BFS dan DFS yang telah kami buat dapat digunakan untuk melakukan pencarian *file* secara *first-occurrence* dan *all-occurence*. Walaupun begitu, pengujian dan pengembangan lebih lanjut tetap perlu dilakukan untuk memaksimalkan kualitas aplikasi ini.

B. SARAN

Salah satu kendala yang kami temui dalam pengembangan aplikasi ini adalah program tidak bisa menampilkan pohon direktori berukuran besar. Hal ini disebabkan oleh keterbatasan dari area *rendering*. Selain itu, perhitungan waktu berjalannya algoritma DFS belum dapat kami pisahkan dari proses pembentukan pohon direktori. Akibatnya, waktu yang diberikan oleh algoritma DFS cenderung lebih lama dan tidak representatif terhadap proses pencariannya. Pemrogram yang hendak mengembangkan aplikasi ini lebih lanjut hendaknya mempertimbangkan adanya permasalahan ini dan mendalami lebih lanjut penyebab dan solusi-solusi yang mungkin diaplikasikan.

LINK *REPOSITORY* GITHUB

Kode program aplikasi *Folder Crawler* ini dapat diakses pada *link* Github berikut ini:

https://github.com/tastytypist/folder-crawler

DAFTAR PUSTAKA

[1] Levitin, A., 2012. *Introduction to the Design & Analysis of Algorithms*. Essex: Pearson.

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

[3] http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf