

LAPORAN TUGAS BESAR
IF2211 STRATEGI ALGORITMA

PENGAPLIKASIAN ALGORITMA BFS DAN DFS
DALAM IMPLEMENTASI *FOLDER CRAWLING*

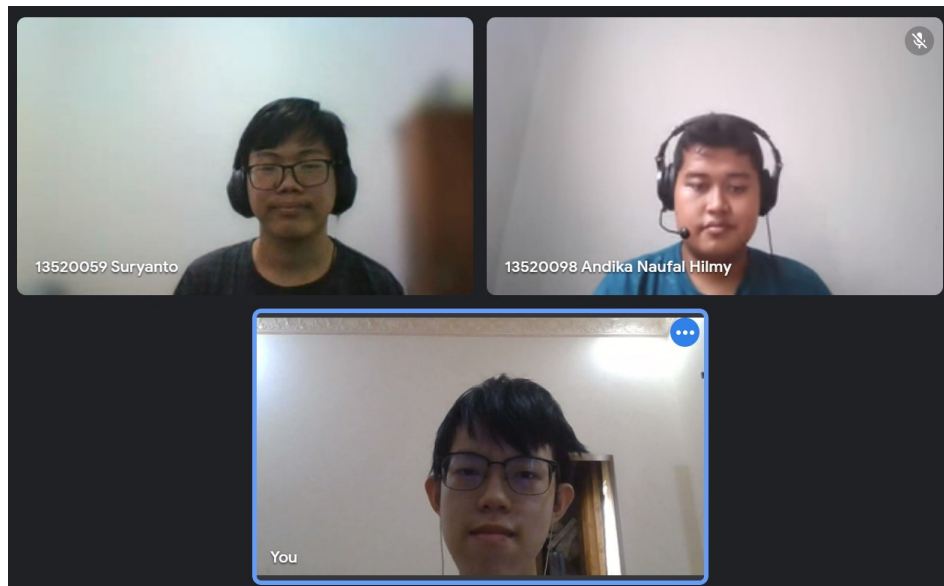
Disusun oleh:

Kelompok filesearch.exe

Suryanto 13520059

Wesly Giovano 13520071

Andika Naufal Hilmy 13520098



TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2022

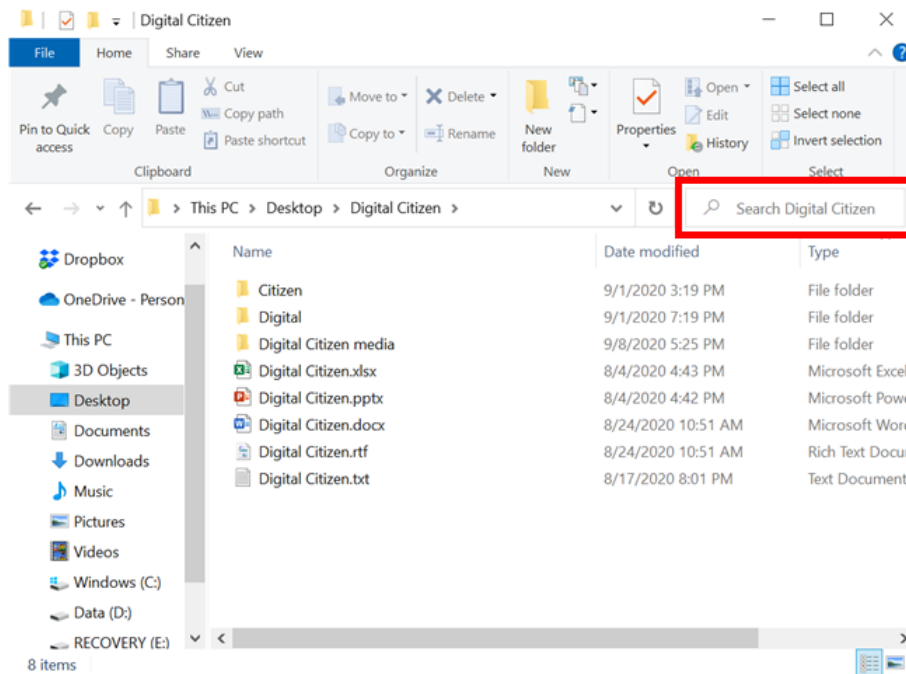
DAFTAR ISI

BAB I DESKRIPSI TUGAS.....	2
1.1 Input dan Output Program.....	3
1.2 Spesifikasi Program	7
1.3 Spesifikasi GUI:	8
BAB II LANDASAN TEORI	9
2.1 Graph Traversal.....	9
2.2 Breadth-First Search.....	9
2.3 Depth-First Search	9
2.4 C# Desktop Application Development	10
BAB III ANALISIS PEMECAHAN MASALAH	11
3.1 Langkah-Langkah Pemecahan Masalah.....	11
3.2 Proses Mapping Persoalan Menjadi Elemen-Element Algoritma BFS dan DFS.....	11
3.3 Ilustrasi Kasus Lain.....	11
BAB IV IMPLEMENTASI DAN PENGUJIAN	13
4.1 Implementasi Program Utama.....	13
4.1.1 Implementasi User Interface	13
4.1.2 Implementasi Breadth-First Search.....	13
4.1.3 Implementasi Depth-First Search.....	14
4.2 Struktur Data	15
4.3 Cara Penggunaan Program.....	15
4.4 Hasil Pengujian	16
4.4.1 Pengujian I	16
4.4.2 Pengujian II.....	17
4.4.3 Pengujian III.....	19
4.4.4 Pengujian IV.....	20
4.5 Analisis Desain Solusi BFS dan DFS	22
BAB V KESIMPULAN DAN SARAN	23
5.1 Kesimpulan	23
5.2 Saran.....	23
REFERENSI.....	24
LAMPIRAN.....	25

BAB I

DESKRIPSI TUGAS

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.

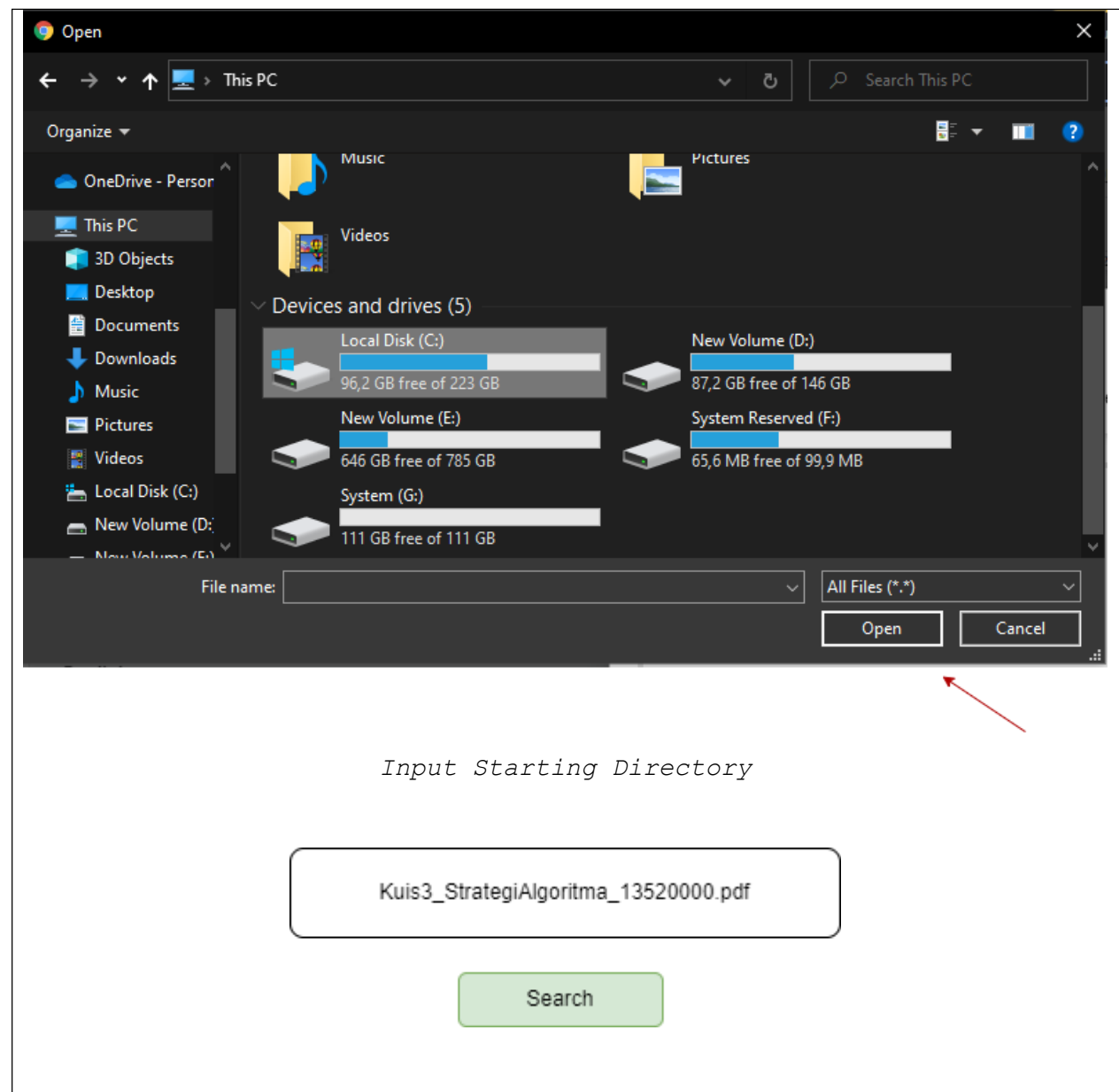
Dalam tugas besar ini, penulis diminta untuk membangun sebuah aplikasi GUI sederhana yang dapat memodelkan fitur dari *file explorer* pada sistem operasi, yang pada tugas ini disebut dengan *Folder Crawling*. Dengan memanfaatkan algoritma *Breadth First Search* (BFS) dan *Depth First Search* (DFS), pengguna dapat

menelusuri folder-folder yang ada pada *directory* untuk mendapatkan *directory* yang diinginkan. Penulis juga diminta untuk memvisualisasikan hasil dari pencarian folder tersebut dalam bentuk pohon.

Selain pohon, Penulis 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.

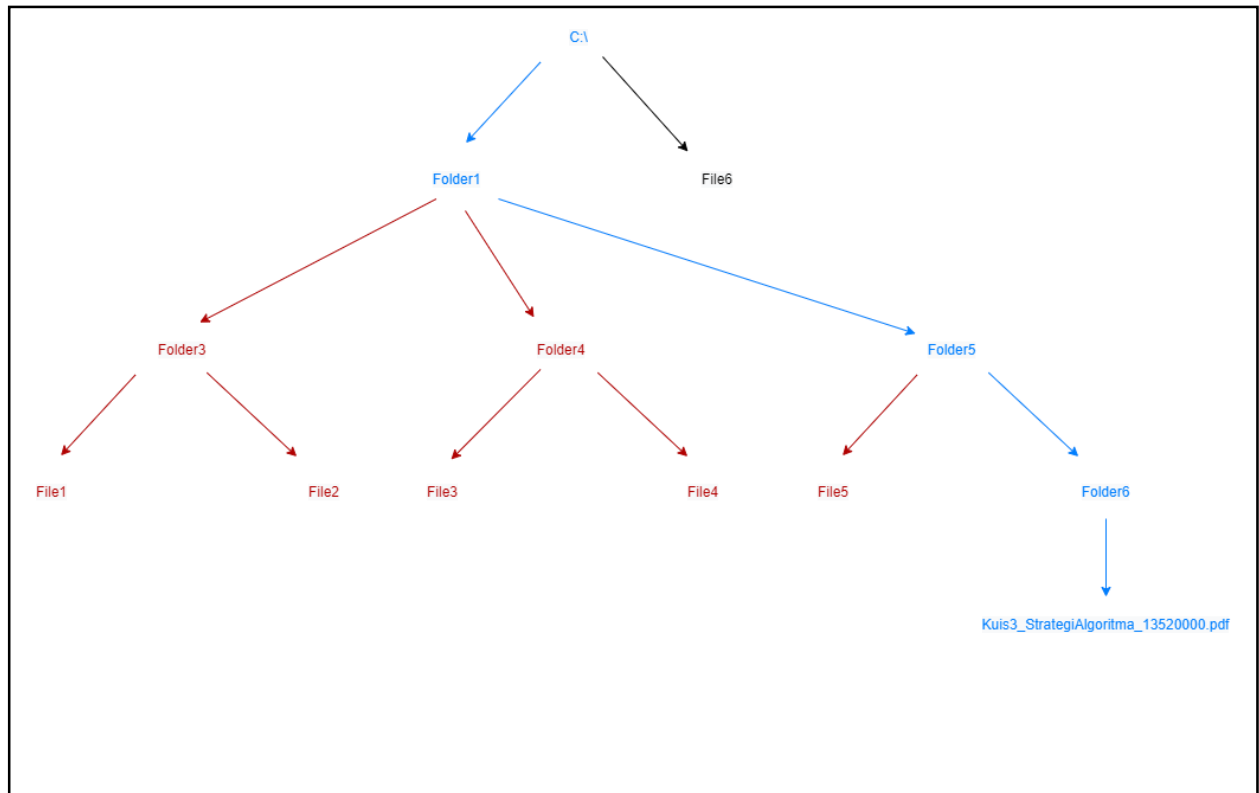
1.1 Input dan Output Program

Contoh masukan aplikasi:



Gambar 2. Contoh input program

Contoh output aplikasi:

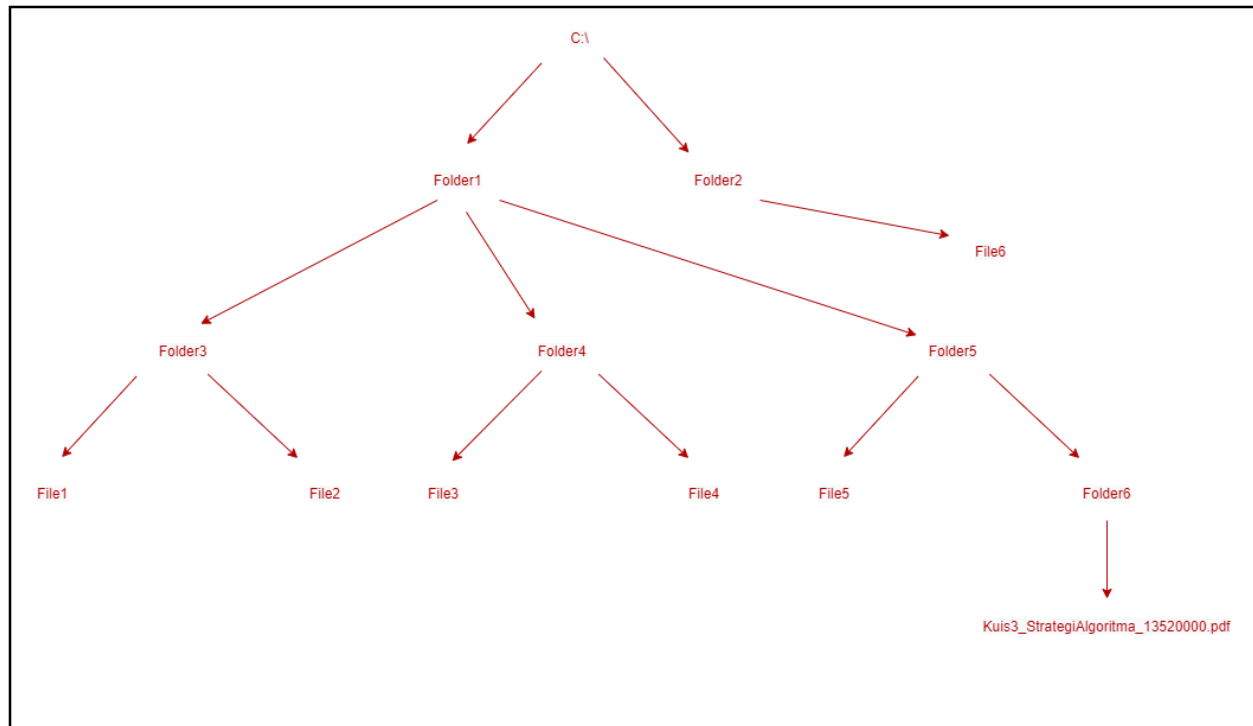


Gambar 3. Contoh output program

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.

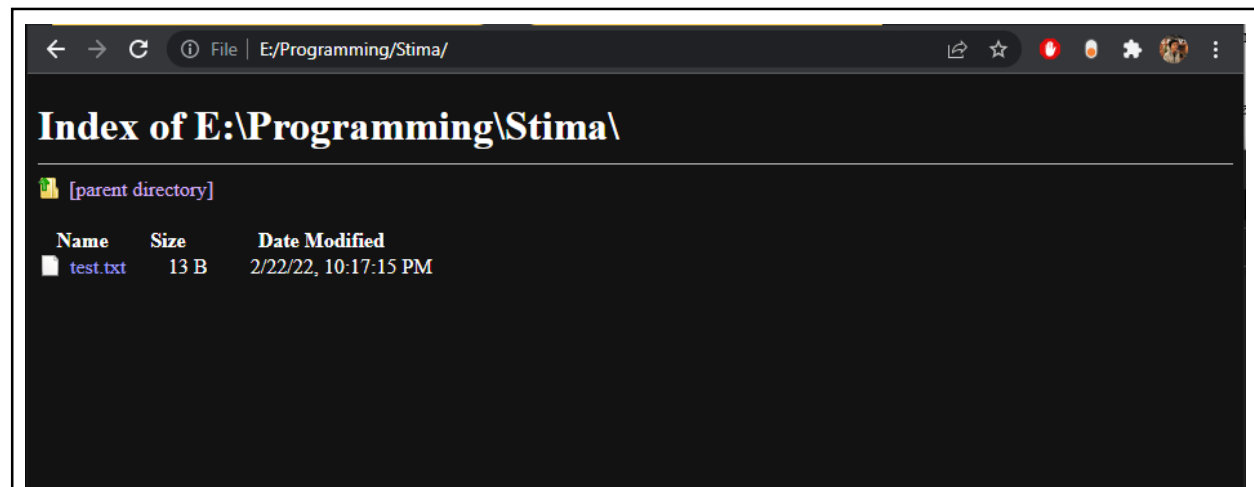


Gambar 4. Contoh output program jika *file* tidak ditemukan

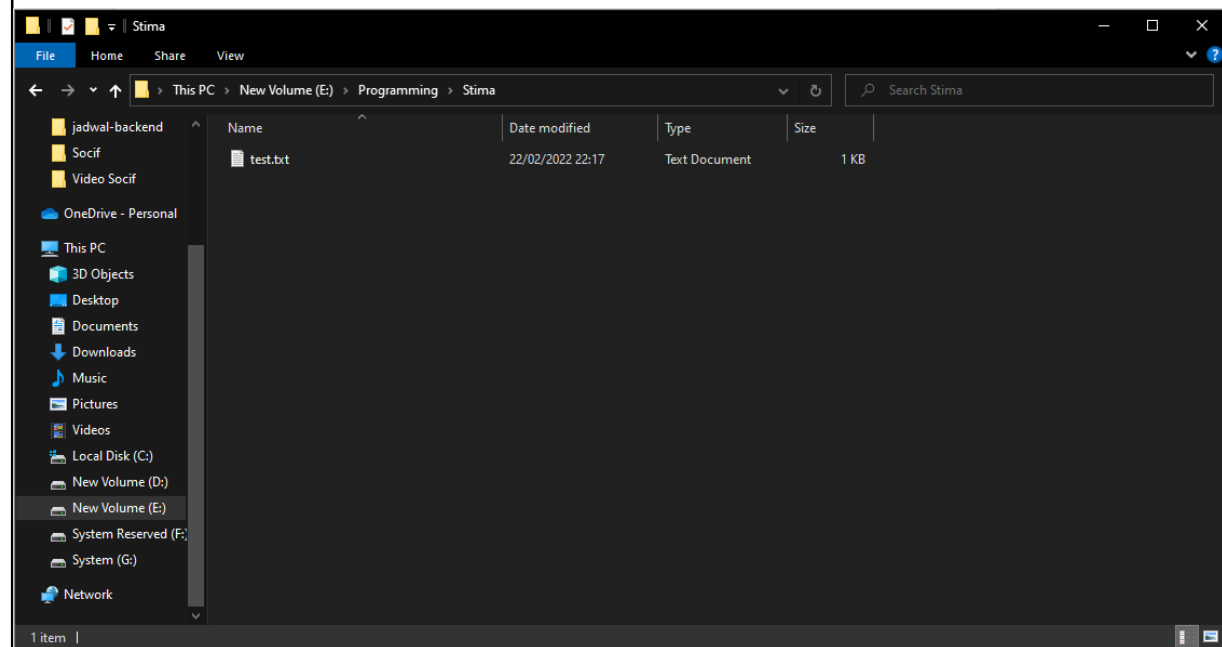
Jika *file* yang ingin dicari pengguna tidak ada pada direktori *file*, misalnya saat pengguna mencari Kuis3Probststat.pdf, maka path pencarian DFS adalah sebagai berikut: C:\ → Folder1 → Folder3 → File1 → Folder3 → File2 → Folder3 → Folder1 → Folder4 → File3 → Folder4 → File4 → Folder4 → Folder1 → Folder5 → 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* :



Contoh Hyperlink Dibuka Melalui Browser



Contoh Hyperlink Dibuka Melalui Browser

Gambar 5. Contoh ketika *hyperlink* di-klik

1.2 Spesifikasi Program

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

Folder Crawling

Input

Choose Starting Directory
[Choose Folder...](#) No File Chosen


Input File Name

☐ Find all occurrence

Input Metode Pencarian
☐ BFS
☒ DFS

[Search](#)

Output



Folder Crawling

Input

Choose Starting Directory
[Change Folder...](#) C:/

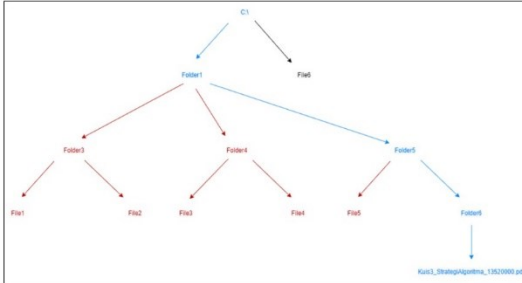
Input File Name

☐ Find all occurrence

Input Metode Pencarian
☐ BFS
☒ DFS

[Search](#)

Output



Path File :
• [C:/Folder1/Folder5/Folder6/Kuis3_StrategiAlgoritma_13520000.pdf](#)

Time spent: 20.02s

Gambar 9. Tampilan layout dari aplikasi desktop yang dibangun

Catatan: Tampilan diatas hanya berupa salah satu contoh layout dari aplikasi saja, untuk designlayout aplikasi dibebaskan dengan syarat mengandung seluruh input dan output yang terdapat pada spesifikasi.

1.3 Spesifikasi GUI

Berikut adalah spesifikasi pada GUI program :

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 II

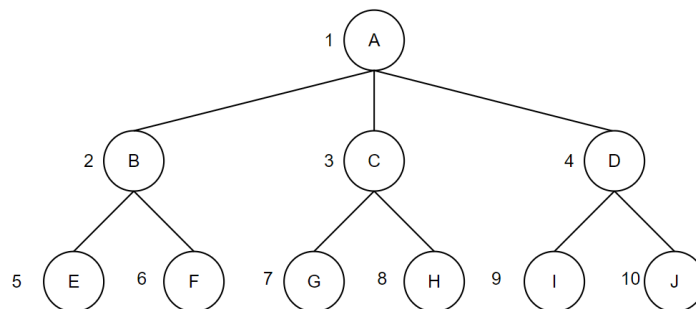
LANDASAN TEORI

2.1 Graph Traversal

Graph traversal adalah suatu proses mengunjungi setiap simpul dalam suatu graf. Proses ini akan menyimpan simpul-simpul yang belum dikunjungi sehingga nantinya dapat dikunjungi meskipun tidak terhubung secara langsung dengan simpul terakhir yang dikunjungi. Terdapat beberapa algoritma yang dapat digunakan untuk menentukan urutan dari pengunjungan simpul yang dilakukan. Algoritma yang berbeda dapat memiliki tingkat keefisienan yang berbeda tergantung dengan kasus yang ditemui. Pada tugas ini, penulis akan mengimplementasikan dua algoritma pada *graph traversal*, yaitu algoritma *Breadth-First Search* (BFS) dan algoritma *Depth-First Search* (DFS).

2.2 Breadth-First Search

Pada algoritma BFS, proses akan dimulai pada suatu simpul awal (*depth* n ; $n = 0, 1, 2, \dots$) dan akan mengunjungi semua simpul tetangga (*depth* $n+1$) dari simpul yang terpilih terlebih dahulu dan akan dilanjutkan ke simpul tetangga dari simpul yang telah dikunjungi (*depth* $n+2$). Umumnya, algoritma ini direpresentasikan dengan struktur data *queue* yang menggunakan prinsip FIFO (*First-In-First-Out*). Simpul-simpul tetangga yang dari simpul yang sedang dikunjungi akan dimasukkan ke dalam *queue* (*enqueue*) apabila sebelumnya belum pernah dikunjungi dalam proses *traversal*. Selanjutnya, elemen pertama dari *queue* akan dikeluarkan (*dequeue*) dan berperan sebagai simpul yang dikunjungi. Proses ini akan terus dilakukan hingga *queue* kosong (semua simpul telah dikunjungi) atau tujuan dari *traversal* telah tercapai.

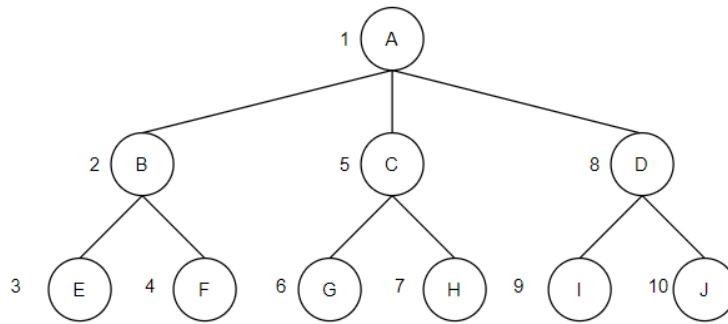


Gambar 10. Ilustrasi Pencarian dengan Algoritma BFS

Urutan pencarian pada gambar diatas : $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G \rightarrow H \rightarrow I \rightarrow J$.

2.3 Depth-First Search

Pada Algoritma DFS, proses akan dimulai dari suatu simpul awal (*depth* 0) yang dan pencarian akan diteruskan ke simpul tetangga (*depth* 1). Sebelum melakukan penelusiran ke simpul lain(*depth* 1), pencarian akan dilakukan pada simpul turunan terlebih dahulu (*depth* 2) hingga *depth* n . Apabila simpul sudah tidak memiliki anak lagi (daun), maka penelusuran akan dilakukan ke simpul dengan *depth* $n+1$.



Gambar 11. Ilustrasi Pencarian dengan Algoritma DFS

Urutan pencarian pada gambar diatas : $A \rightarrow B \rightarrow E \rightarrow F \rightarrow C \rightarrow G \rightarrow H \rightarrow D \rightarrow I \rightarrow J$.

2.4 C# Desktop Application Development

C# adalah salah satu bahasa pemrograman yang mendukung pengembangan aplikasi berbasis desktop bersama dengan framework bawaannya yaitu .NET Framework, meskipun sekarang ini telah digantikan oleh framework .NET Core. Secara umum, pengembangan aplikasi berbasis desktop dengan C# hanya efektif dan efisien untuk operating system Windows. C# memiliki IDE bawaan yaitu Visual Studio, sehingga pada umumnya pengembangan aplikasi dengan C# ditulis pada Visual Studio, meskipun ada IDE lain yang juga mendukung pengembangan ini misalnya MonoDevelop dan JetBrains Rider. Ada beberapa *framework approach* di .NET Framework, antara lain WinForms (Windows Form Application) dan WPF (Windows Presentation Foundation). WinForms merupakan framework yang tidak memerlukan *markup language* dalam desain UI-nya karena GUI yang dimiliki WinForms terbentuk berdasarkan *event-driven control* yang telah didefinisikan, sedangkan WPF menggunakan *markup language* untuk didesain sendiri UI yang diinginkan.

BAB III

ANALISIS PEMECAHAN MASALAH

3.1 Langkah-Langkah Pemecahan Masalah

Berikut adalah langkah-langkah yang dilakukan oleh penulis dalam proses pengerjaan tugas ini.

1. Pembuatan algoritma DFS dan BFS untuk pencarian *file* di suatu *directory*
2. Pembuatan Kelas *DrawingTree* untuk membantu dalam proses penggambaran graf yang digambarkan dalam bentuk pohon
3. Pembuatan main program untuk menguji algoritma yang telah dibuat
4. Integrasi program pencarian dengan GUI
 - Pada bagian ini main program dipindahkan ke dalam program dalam bentuk WPF (Windows Presentation Foundation) yang digabungkan dengan algoritma *file traversal* yang telah dibuat.

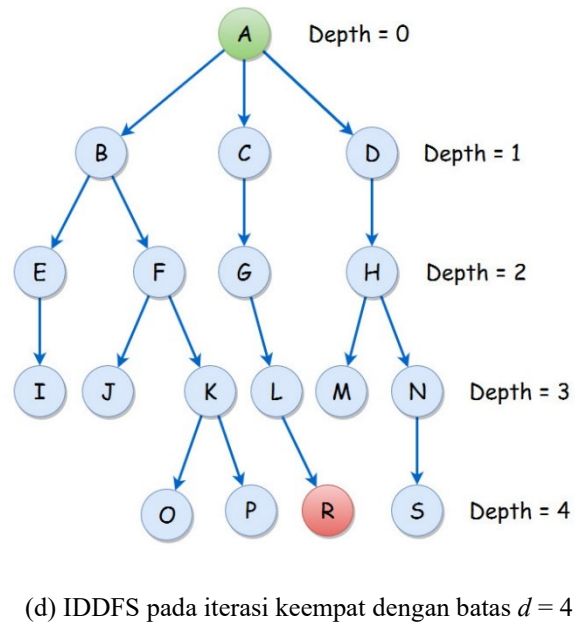
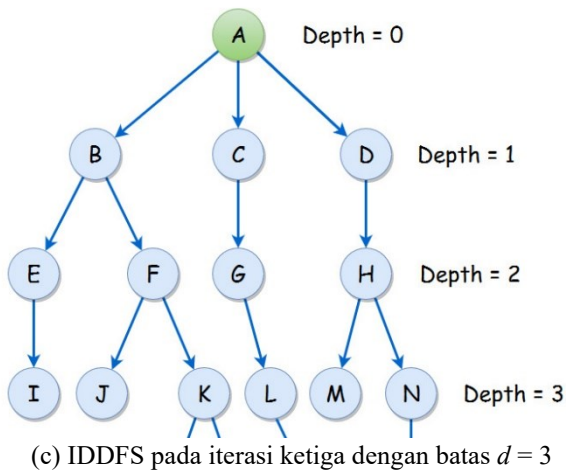
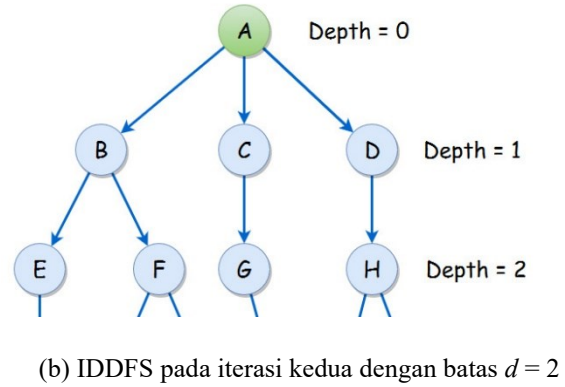
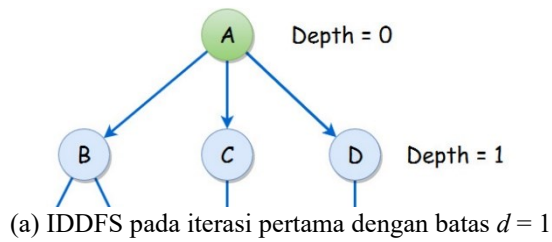
3.2 Proses Mapping Persoalan Menjadi Elemen-Elemen Algoritma BFS dan DFS

Berikut adalah proses mapping yang dilakukan.

1. *Root directory* yang dipilih oleh user dijadikan sebagai simpul akar
2. Setiap *file* dan subfolder dari sebuah folder direpresentasikan sebagai simpul anak.
3. Pencarian dilakukan dengan mengunjungi *file* terlebih dahulu agar program pencarian lebih efisien.
4. *File* yang ingin dicari akan menjadi *goal state* pada proses pencarian.

3.3 Ilustrasi Kasus Lain

Pemecahan masalah pencarian *file* dalam direktori dapat menggunakan pendekatan IDDFS (iterative deepening depth-first search), yaitu pendekatan DLS (depth-limited search) yang batas kedalamannya di-*increment* jika tidak berhasil ditemukan. Pendekatan ini bersifat optimal, meskipun waktu yang dibutuhkan lebih lama. Misalnya, pada iterasi pertama dilakukan DLS dengan batas kedalaman 1. Jika tidak ditemukan, maka akan dilakukan DLS dengan batas kedalaman 2, dan seterusnya hingga *file* ditemukan. Kompleksitas waktu dari IDDFS sama dengan pendekatan BFS, yaitu $O(b^d)$, dan kompleksitas ruangnya adalah $O(d)$. Akibatnya, pendekatan IDDFS ini cocok untuk pencarian dengan ruang pencarian yang besar dan kedalaman yang tidak diketahui; secara spesifik, pendekatan IDDFS cocok untuk pencarian *file* pula.



Gambar 12. Ilustrasi pendekatan IDDFS dengan pencarian dimulai dari simpul A terhadap simpul R

BAB IV IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Program Utama

4.1.1 Implementasi User Interface

```
{ Asumsi: Semua variable adalah global variable }

if isClicked(startingDirectoryButton) then
    open folderPicker

if isClicked(searchButton) then
    if isEmpty(startingDirectory) then
        print(error)
    if isEmpty(fileName) then
        print(error)
    if isChecked(bfsButton) and isChecked(DFSButton) then
        print(error)

    t ← DrawingTree()

    if isChecked(bfsButton) then
        BFS(startingDirectory, fileName, isChecked(allOccurrence), t)
    else
        DFS(startingDirectory, fileName, isChecked(allOccurrence), t, false)

    display(t)
    display(Solution)
```

4.1.2 Implementasi Breadth-First Search

Berikut adalah *pseudocode* dari algoritma BFS yang digunakan.

```
Function BFS (root : string, searchValue: string, allOccurrence : boolean, t:
drawingTree) → Boolean

KAMUS
Solution                : list of string
files, subDirs          : array of string
found                   : boolean
dirQueue                : queue of string
currentDir, filename, dirname : string

{Mengembalikan array of string yang merupakan nama file di dalam folder dir}
function getFiles(dir : string) → array of string
```

```

{Mengembalikan array of string yang merupakan nama folder di dalam folder dir}
function getDirectory(dir : string) → array of string

ALGORITMA
DirQueue.enqueue(root)
while (not dirQueue.isEmpty and (not found or allOccurrence)):
    currentDir ← dirQueue.dequeue()
    files ← getFiles(currentDir)
    subDirs ← getDirectory(currentDir)

    foreach (fileName in files):
        if fileName = searchValue:
            Solution ← Solution + fileName
            found ← true
            If (!allOccurrence):
                Break
            Endif
        endif
    endfor

    foreach (subDirName in subDirs):
        if (not found or allOccurrence):
            DirQueue.enqueue(subDirName)
        endif
    endfor
→ found

```

4.1.3 Implementasi Depth-First Search

Berikut adalah *pseudocode* dari algoritma DFS yang digunakan.

```

Function DFS (root : string, searchValue : string, allOccurrence : boolean,
t : drawingTree, found : Boolean) → Boolean

KAMUS
Solution                : list of string
files, subDirs          : array of string
subDirName, fileName    : string
t1                      : drawingTree          {class untuk membentuk tree}

{Mengembalikan array of string yang merupakan nama file di dalam folder dir}
function getFiles(dir : string) → array of string

{Mengembalikan array of string yang merupakan nama folder di dalam folder dir}
function getDirectory(dir : string) → array of string

ALGORITMA
files ← getFiles(root)
subDirs ← getDirectory(root)

foreach (fileName in files):
    if fileName = searchValue:

```

```

        Solution ← Solution + fileName
        found ← true
        If (!allOccurrence):
            Break
        Endif
    endif
endfor

foreach (subDirName in subDirs):
    if (not found or allOccurrence):
        found ← DFS(subDirName, searchValue, allOccurrence, t1, found)
    endif
endfor
→ found

```

4.2 Struktur Data

Struktur data yang digunakan dalam program ini yaitu sebuah kelas `DrawingTree` dengan atribut

- `graph`, bertipe `Microsoft.Msagl.Drawing.Graph` untuk menyimpan pohon yang telah terbentuk,
- `rootId`, bertipe `string` untuk menyimpan id simpul akar dari `graph`, dan
- `nodeCount`, bertipe `int` dan bersifat `static` untuk mencatat jumlah simpul yang telah terbentuk agar setiap id simpul yang akan terbentuk dapat dipastikan unik.

Di sisi lain, kelas `DrawingTree` juga memiliki beberapa method, yaitu

- getter dari atribut `graph` dan `rootId`,
- konstruktor `DrawingTree()` untuk membentuk pohon kosong dan `DrawingTree(name, color)` untuk membentuk pohon dengan satu simpul akar,
- `AddChild()` untuk menambahkan simpul anak pada simpul akar,
- `UpdateColor()` yang bersifat `private` untuk meng-*update* warna orangtua berdasarkan simpul anak yang ditambahkan dengan `AddChild()`, dan
- `SetColor()` untuk mengubah warna sebuah simpul tertentu.

Struktur data lain yang digunakan berasal dari *package* yang digunakan, antara lain kelas `Graph` dan `Color` dari *package* `MSAGL`.

4.3 Cara Penggunaan Program

Berikut adalah langkah-langkah menggunakan program.

1. Pilih direktori yang diinginkan sebagai direktori akar pencarian *file* dengan tombol “Choose”.
2. Masukkan nama *file* yang hendak dicari termasuk *extension*-nya, misalnya “file1.txt”.
3. Centang *checkbox* “Find all occurrences” bila ingin mencari semua *file* dengan nama yang telah ditentukan (bersifat opsional).
4. Pilih metode yang diinginkan dalam pencarian *file*, yaitu dengan metode BFS atau DFS.

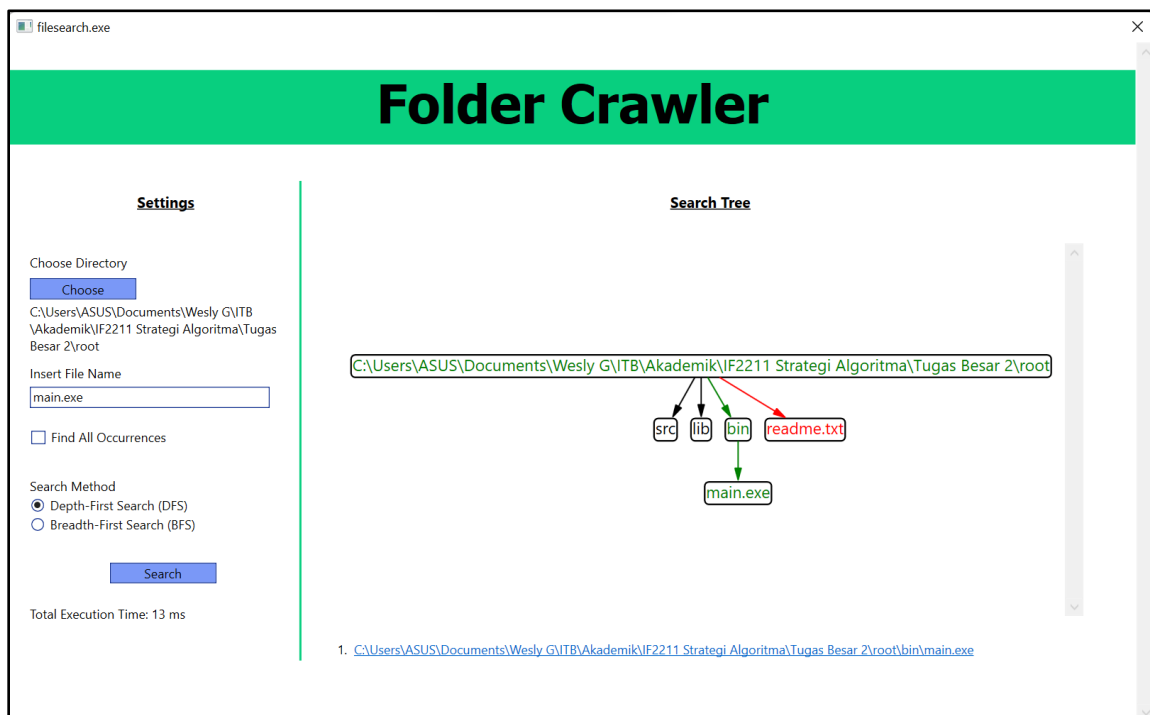
5. Tekan tombol “Search” untuk mencari dan pohon pencarian akan ditampilkan beserta *hyperlink* dari masing-masing *file* yang berhasil dicari menuju ke *parent directory*-nya.

4.4 Hasil Pengujian

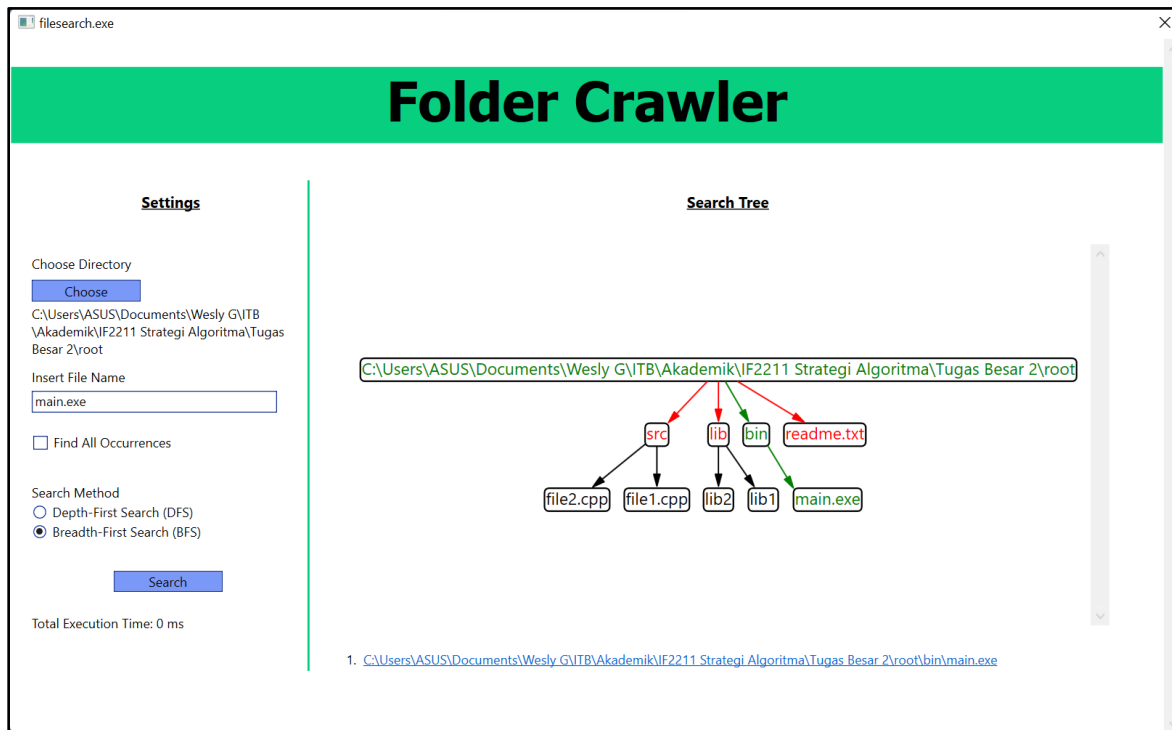
4.4.1 Pengujian I

Dilakukan pencarian terhadap *file* `main.exe` pada folder `root` dengan struktur direktori sebagai berikut.

```
root/  
├─ bin/  
│   └─ main.exe  
├─ lib/  
│   ├── lib1/  
│   └── lib2/  
│       ├── file1.cpp  
│       └── somelib.dll  
├─ src/  
│   ├── file1.cpp  
│   └── file2.cpp  
└─ readme.txt
```



Gambar 13. Pencarian I dengan metode DFS

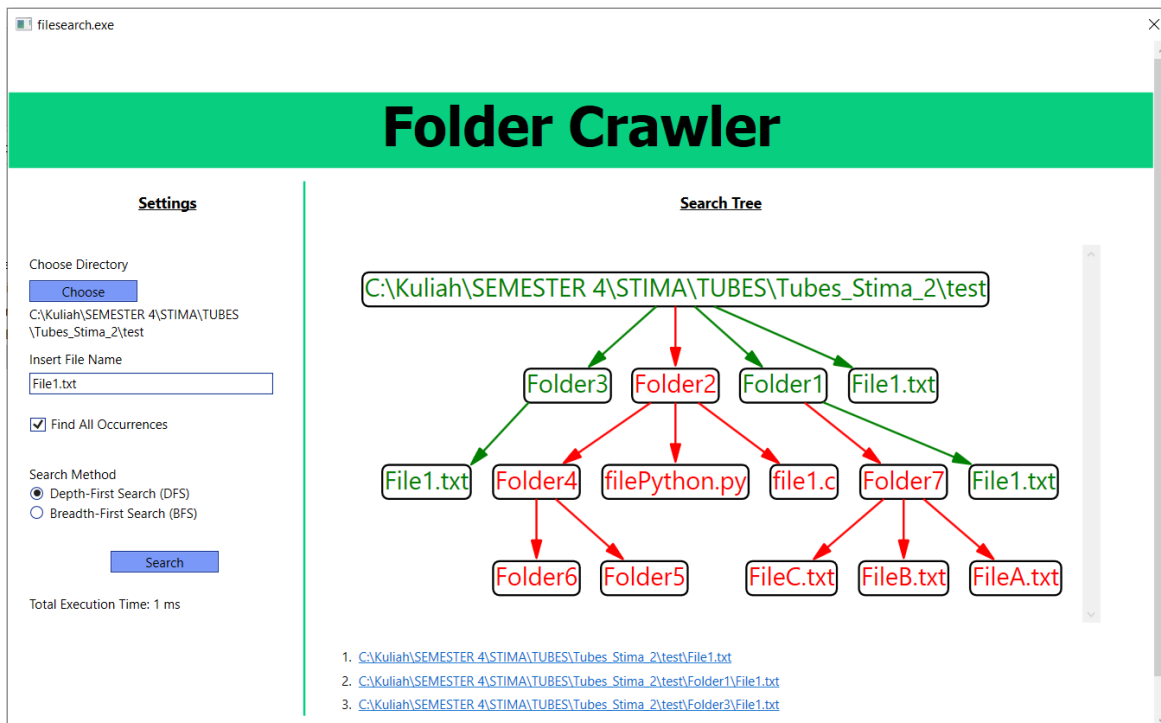


Gambar 14. Pencarian I dengan metode BFS

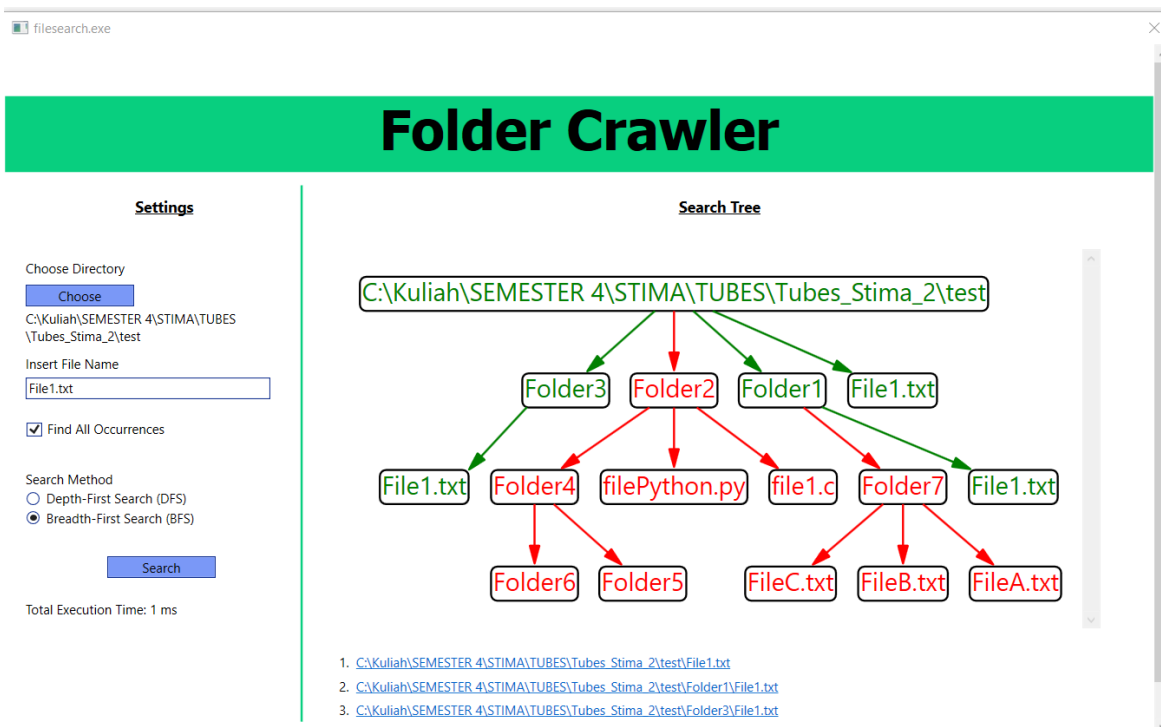
4.4.2 Pengujian II

Dilakukan pencarian terhadap *file* `File1.txt` pada folder `test` dengan struktur direktori sebagai berikut.

```
test/
├─ Folder1/
│   └─ Folder7/
│       ├── FileA.txt
│       ├── FileB.txt
│       └─ FileC.txt
│   └─ File1.txt
├─ Folder2/
│   └─ Folder4/
│       ├── Folder5/
│       └─ Folder6/
│   └─ filePython.py
│   └─ file1.c
├─ Folder3/
│   └─ File1.txt
```



Gambar 15. Pencarian II dengan metode DFS

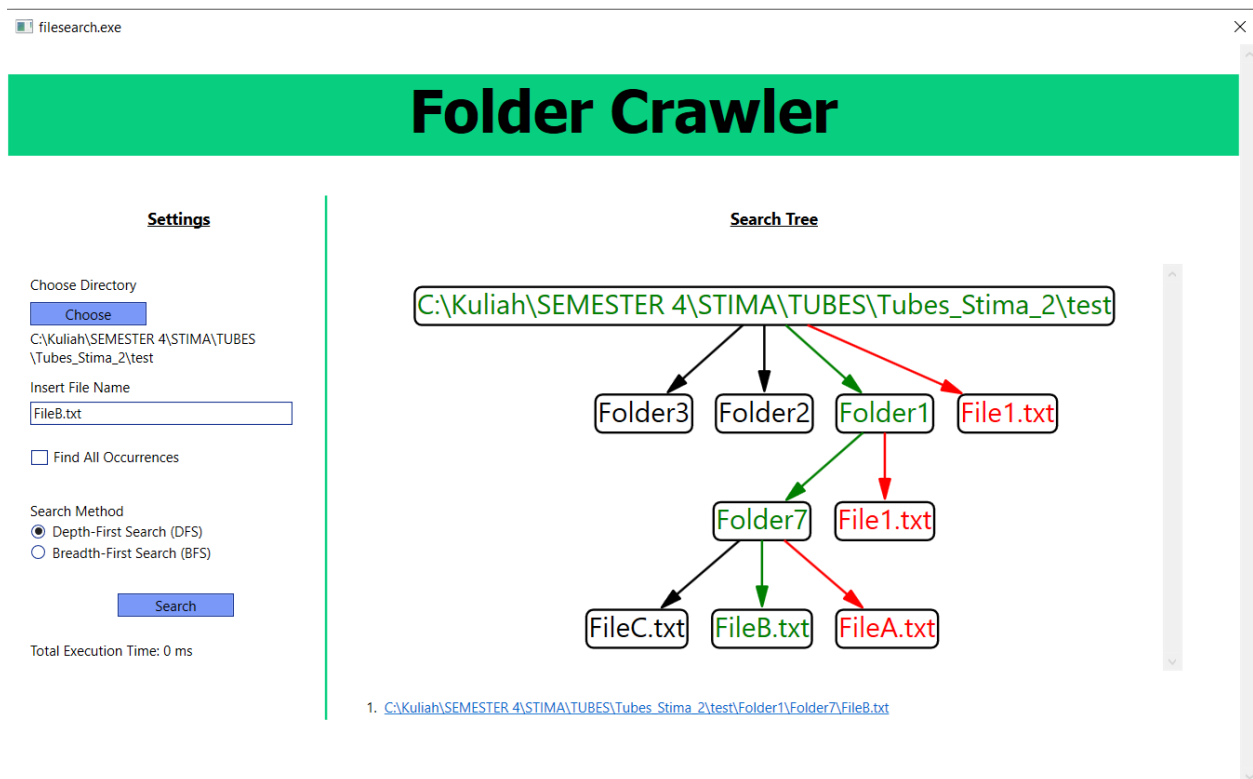


Gambar 16. Pencarian II dengan metode BFS

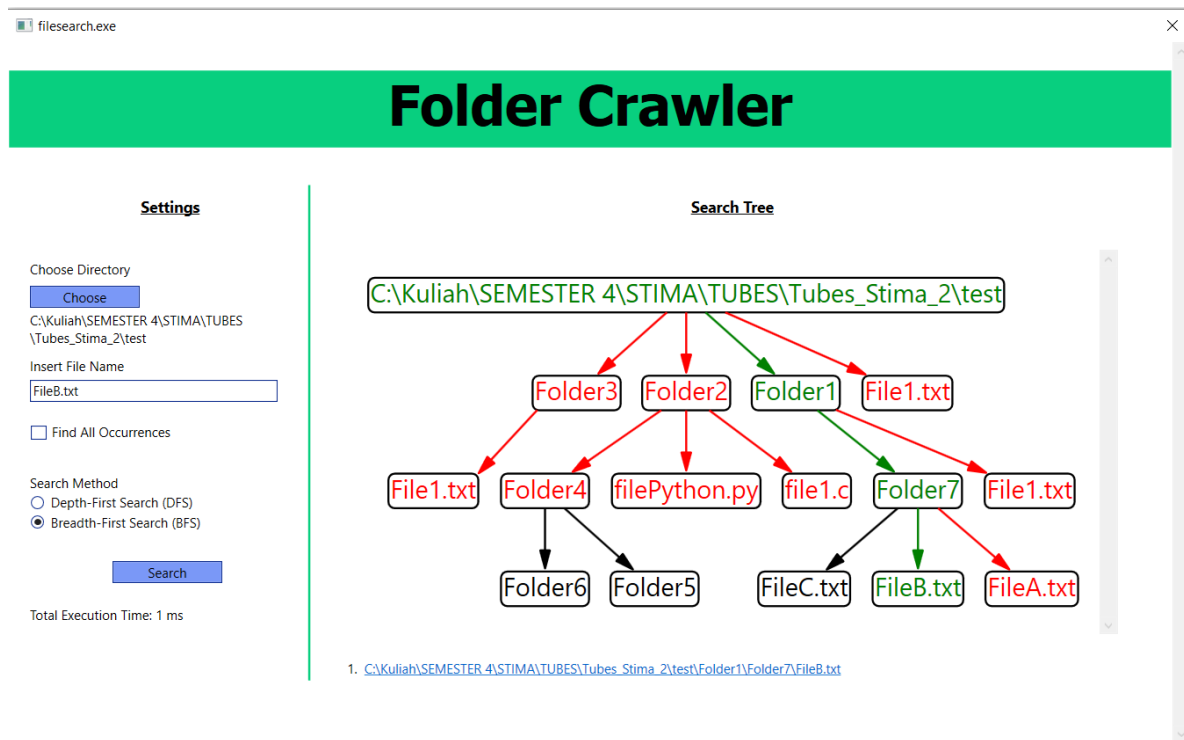
4.4.3 Pengujian III

Dilakukan pencarian terhadap *file* FileB.txt pada folder test dengan struktur direktori sebagai berikut.

```
test/  
├─ Folder1/  
│   ├─ Folder7/  
│   │   ├─ FileA.txt  
│   │   ├─ FileB.txt  
│   │   └─ FileC.txt  
│   └─ File1.txt  
├─ Folder2/  
│   ├─ Folder4/  
│   │   └─ Folder5/  
│   │       └─ Folder6/  
│   └─ filePython.py  
│   └─ file1.c  
├─ Folder3/  
└─ File1.txt
```



Gambar 17. Pencarian III dengan metode DFS

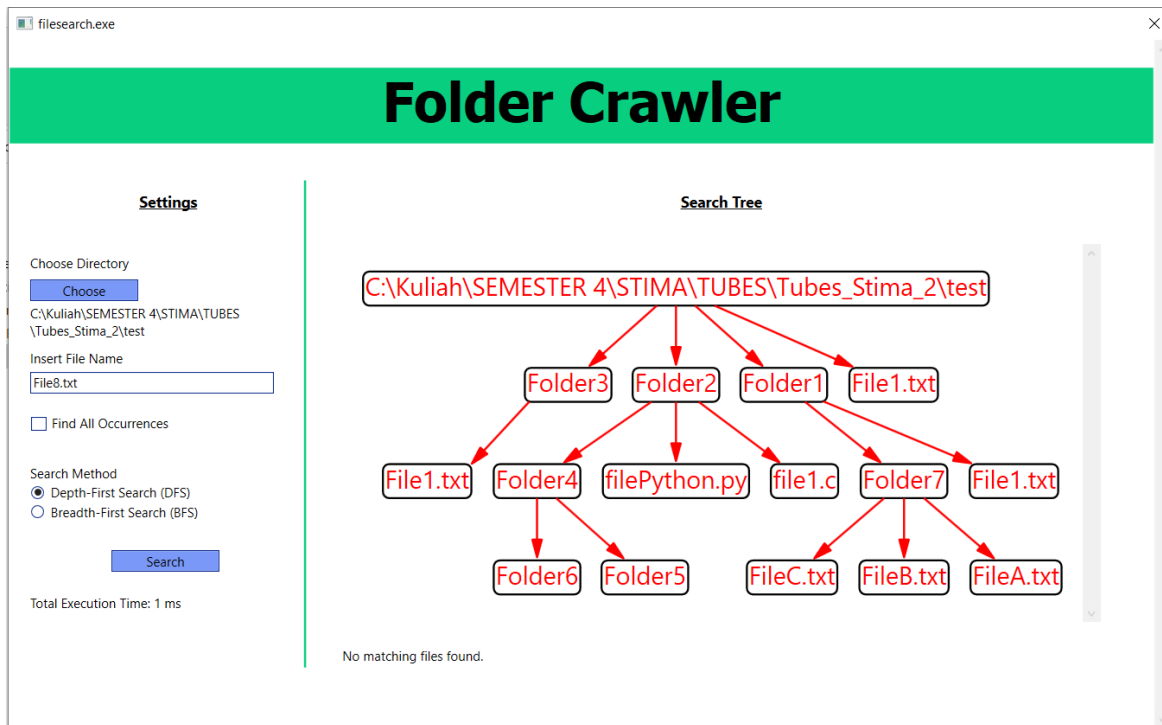


Gambar 18. Pencarian III dengan metode BFS

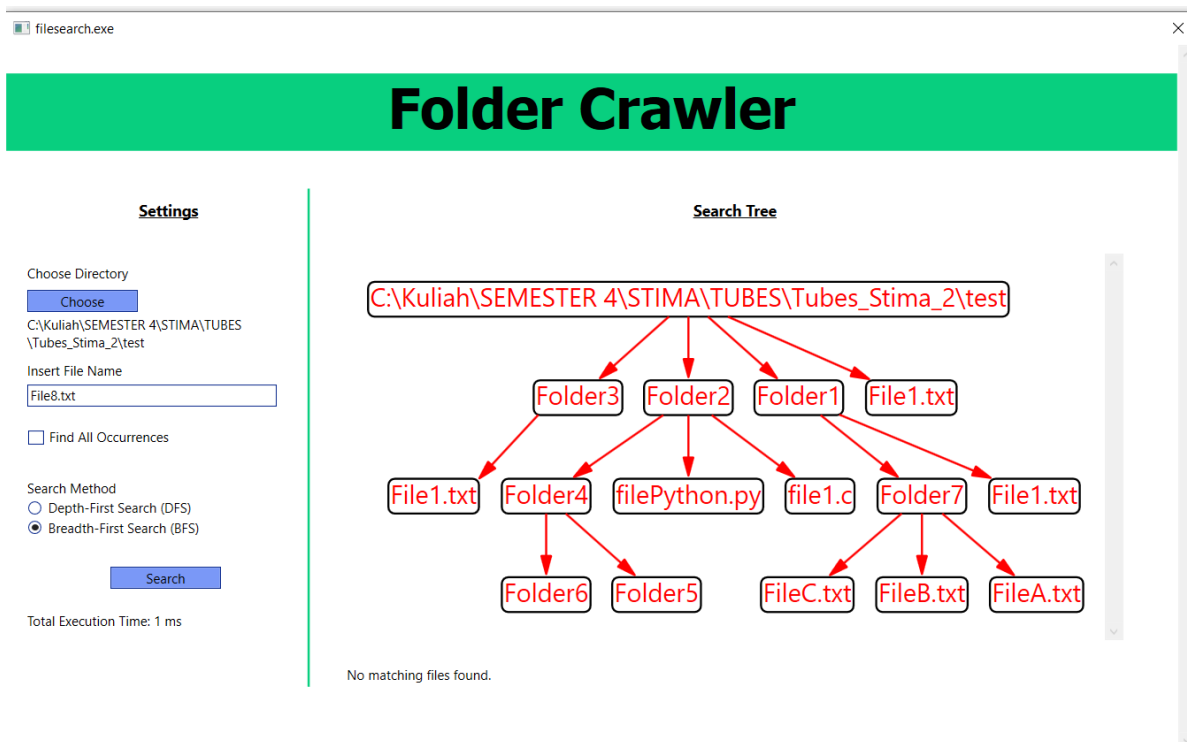
4.4.4 Pengujian IV

Dilakukan pencarian terhadap *file* File8.txt pada folder test dengan struktur direktori sebagai berikut.

```
test/
├─ Folder1/
│   └─ Folder7/
│       ├── FileA.txt
│       ├── FileB.txt
│       └─ FileC.txt
│   └─ File1.txt
├─ Folder2/
│   └─ Folder4/
│       ├── Folder5/
│       └─ Folder6/
│   └─ filePython.py
│   └─ file1.c
├─ Folder3/
└─ File1.txt
```



Gambar 19. Pencarian IV dengan metode DFS



Gambar 20. Pencarian IV dengan metode BFS

4.5 Analisis Desain Solusi BFS dan DFS

Kedua algoritma dapat digunakan untuk melakukan pencarian *file* pada suatu *directory*. Kedua algoritma akan berperan lebih baik tergantung dari kasus yang sedang diuji, yang dalam hal ini berarti tergantung dari struktur *file* dan folder dari *directory* yang ingin diperiksa.

Pada kasus pencarian terhadap *file* yang tidak ada di dalam *directory*, maka kedua algoritma akan memiliki *time complexity* dan *space complexity* yang sama karena semua simpul (*file* dan folder) akan dikunjungi. Hal ini juga berlaku pada kasus pencarian *all occurrences*.

Selanjutnya, apabila struktur dari *file* seperti pada pengujian II, III, dan IV maka metode yang lebih baik tergantung pada *file* yang ingin dicari. Misalnya pencarian *FileA.txt*, maka metode DFS akan lebih baik karena hanya akan melakukan pencarian ke 5 simpul saja dibandingkan dengan BFS yang harus melakukan pencarian ke 11 simpul. Namun, apabila *file* yang ingin dicari adalah *file1.c*, maka BFS akan lebih unggul karena hanya mengunjungi 7 simpul saja, sedangkan DFS melakukan pemeriksaan ke 9 simpul.

Secara keseluruhan, metode DFS akan lebih unggul apabila struktur *directory* memiliki banyak folder dan subfolder yang dalam (banyak folder dibanyak folder). Sedangkan metode BFS akan lebih baik untuk kasus folder hanya memiliki sedikit subfolder.

BAB V

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Program pencarian *file* berhasil diimplementasikan dalam C# Desktop Application dengan menggunakan .NET Framework. Pengguna dapat menginput direktori akar dan nama *file* yang akan dicari beserta metode pencariannya, dan program akan memberikan output berupa pohon pencarian dan *hyperlink* menuju *parent directory* dari *file* yang berhasil dicari.

5.2 Saran

Penulis menyarankan pembaca untuk melakukan lebih banyak terkait algoritma BFS dan DFS sehingga dapat menuliskan algoritma yang lebih cepat, tepat, dan rapi. Eksplorasi terkait pengembangan aplikasi *desktop* dengan bahasa pemrograman C# juga harus dilakukan agar dapat membangun aplikasi yang lebih baik, baik dalam segi GUI maupun fungsionalitas. Salah satu fungsionalitas yang belum dapat dikembangkan oleh penulis adalah tampilan proses pembentukan pohon secara *real time* pada saat pemeriksaan folder/*file* yang sedang berlangsung.

REFERENSI

1. Microsoft. “How to iterate through a directory tree”. Diakses pada Minggu, 13 Maret 2022 melalui <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/file-system/how-to-iterate-through-a-directory-tree>
2. Munir, Rinaldi. “Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1) (Versi Baru 2021)” . Diakses pada Minggu, 13 Maret 2022 melalui <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>
3. Munir, Rinaldi. “Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 2) (Versi Baru 2021)” . Diakses pada Minggu, 13 Maret 2022 melalui <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf>
4. Sangam, Vamsi. “Iterative Deepening Depth First Search (IDDFS)”. Diakses pada 25 Maret 2022 melalui <http://theoryofprogramming.com/2018/01/14/iterative-deepening-depth-first-search-iddfs/>

LAMPIRAN

Link Repository : https://github.com/SurTan02/Tubes_Stima_2

Link Video : <https://youtu.be/5Gx1qt0lAbw>