LAPORAN TUGAS BESAR II
IF2211 STRATEGI ALGORITMA

PENGAPLIKASIAN ALGORITMA BFS DAN DFS DALAM IMPLEMENTASI *FOLDER CRAWLING*

Disusun oleh:

Kelompok PenjelajahHatiAyank (31)

Ignasius Ferry Priguna	13520126
Nelsen Putra	13520130
Daffa Romyz Aufa	13520162

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

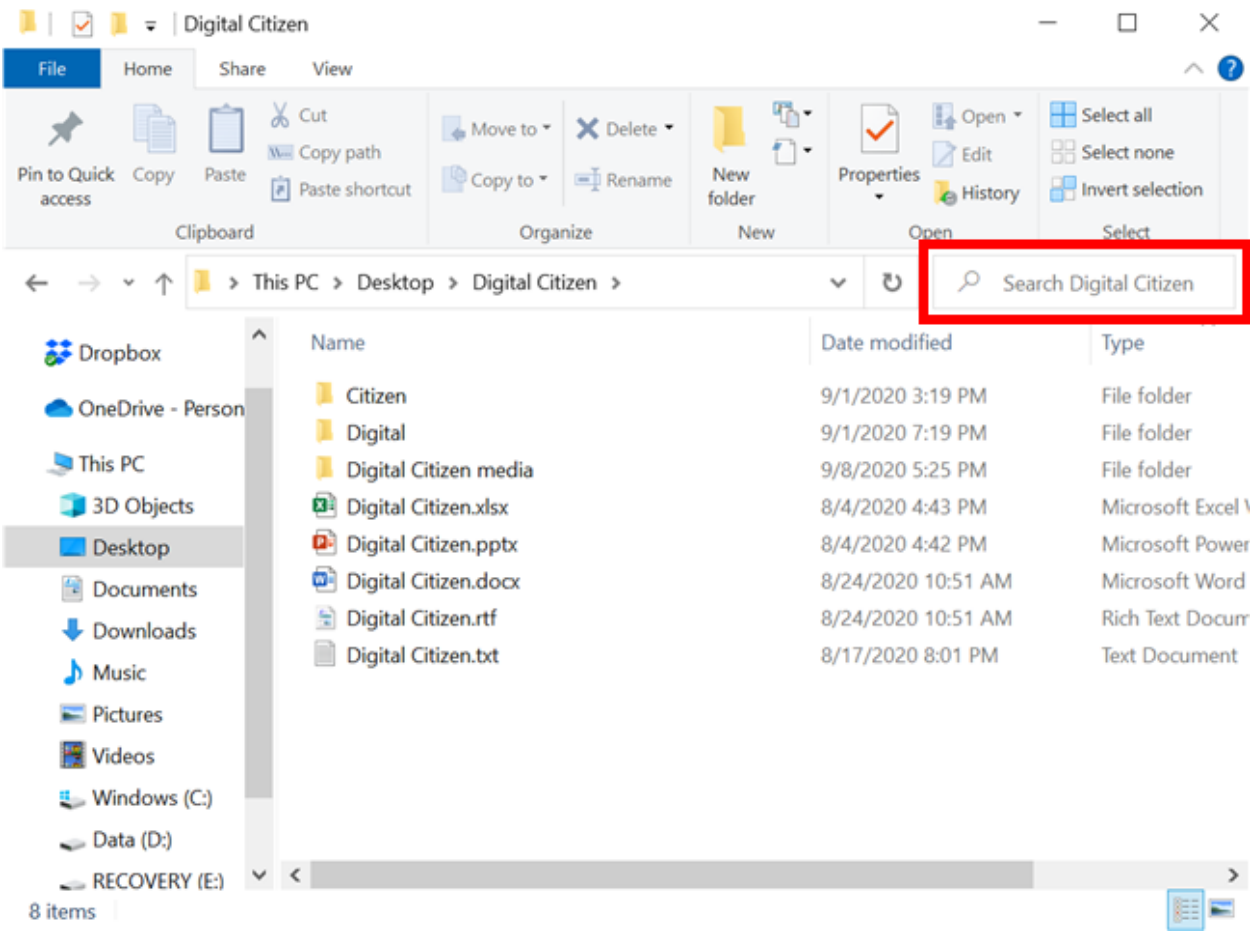
2022

DAFTAR ISI

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.



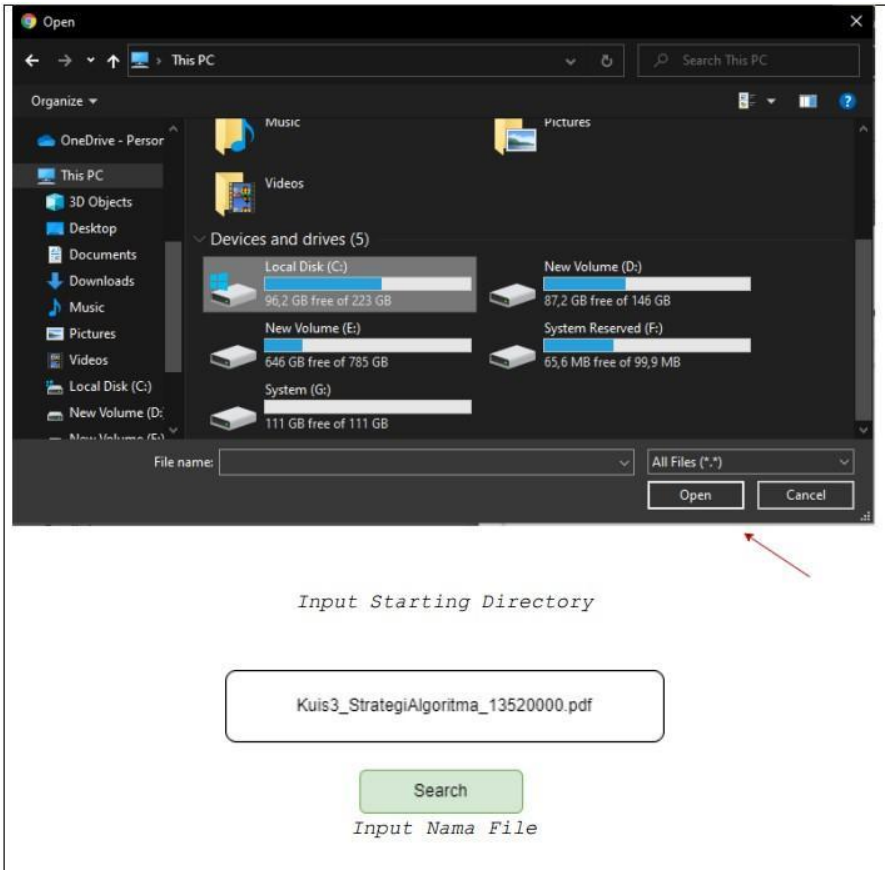
Gambar 1.1 Fitur *Search* pada *Windows 10 File Explorer*

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.

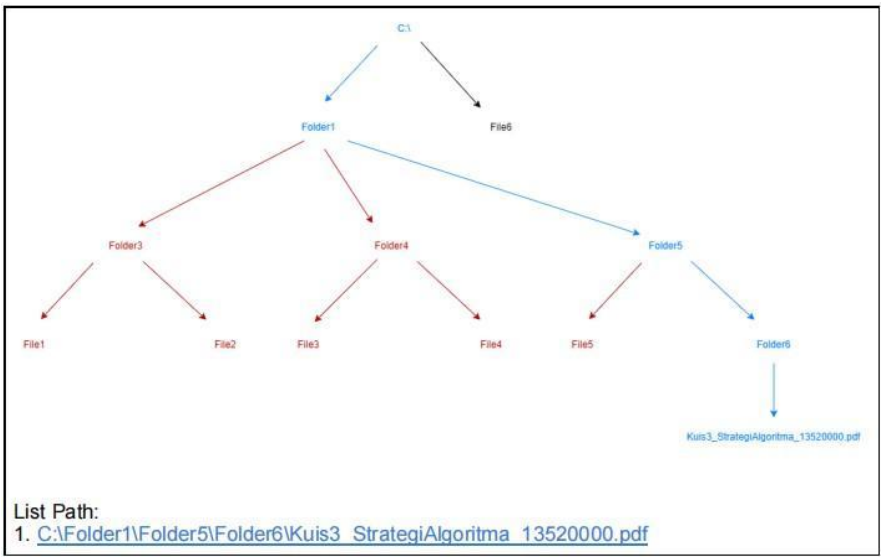
Dalam tugas besar ini, setiap kelompok diminta untuk membangun sebuah aplikasi GUI sederhana yang dapat memodelkan fitur dari *file explorer* pada sistem operasi, yang pada tugas ini disebut dengan *Folder Crawling*. Dengan memanfaatkan algoritma *Breadth First Search* (BFS) dan *Depth First Search*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

Selain pohon, aplikasi yang dibuat juga harus bisa 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.



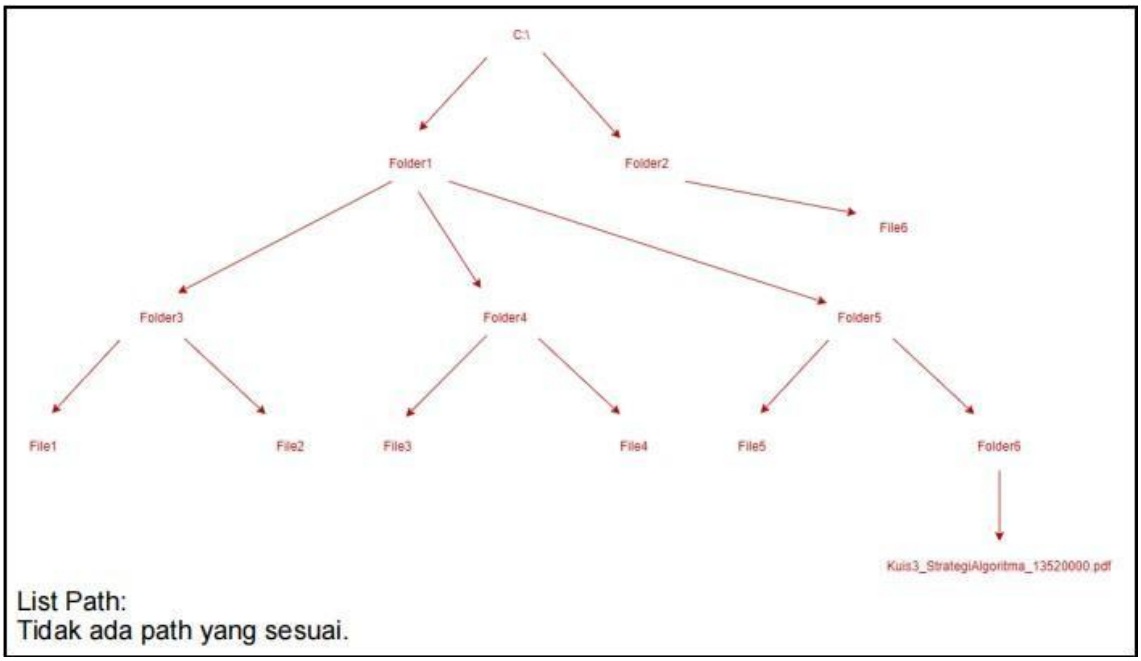
Gambar 1.2 Contoh *Input* Program

Contoh output aplikasi :



Gambar 1.3 Contoh *Output* Program

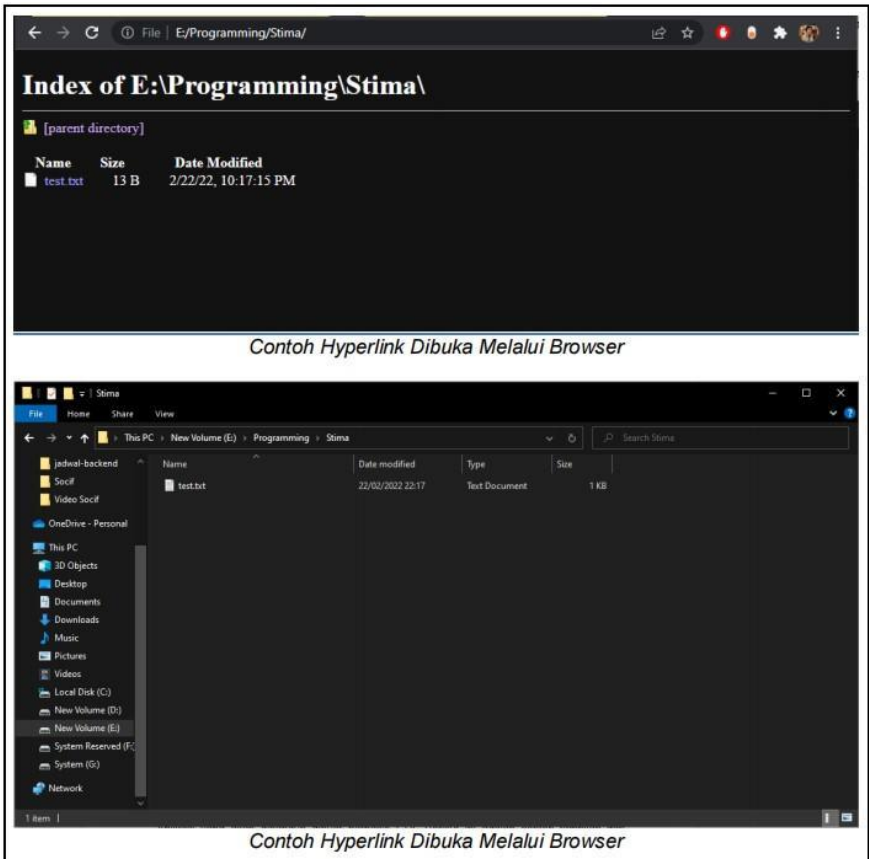
Misalnya pengguna ingin mengetahui langkah *folder crawling* untuk menemukan *file* dengan nama Kuis3_StrategiAlgoritma_13520000.pdf. Maka, path pencarian DFS adalah sebagai berikut. C:\ → Folder1 → Folder3 → File1 → Folder3 → File2 → Folder3 → Folder1 → Folder4 → File3 → Folder4 → File4 → Folder4 → 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. Pemilihan warna dibebaskan asalkan dibedakan antara ketiga hal tersebut.



Gambar 1.3 Contoh *Output* Program Jika File Tidak Ditemukan

Jika *file* yang ingin dicari pengguna tidak ada pada direktori *file*, misalnya saat pengguna mencari Kuis3Probstat.pdf, maka *path* pencarian DFS adalah sebagai berikut: C:\ → Folder1 → Folder3 → File1 → Folder3 → File2 → Folder3 → Folder1 → Folder4 → File3 → Folder4 → File4 → Folder4 → Folder1 → Folder5 → File5 → Folder5 → Folder6 → Kuis3_StrategiAlgoritma_13520000.pdf → Folder6 → Folder5 → Folder1 → C:\ → Folder2 → File6. Pada gambar di atas, semua simpul dan cabang berwarna merah yang menandakan seluruh direktori sudah selesai diperiksa semua namun tidak ada yang mengarah ke tempat *file* berada.

Contoh *hyperlink* pada *path*:



Gambar 1.5 Contoh Ketika *Hyperlink* Di-klik

BAB II

LANDASAN TEORI

2.1. Dasar Teori

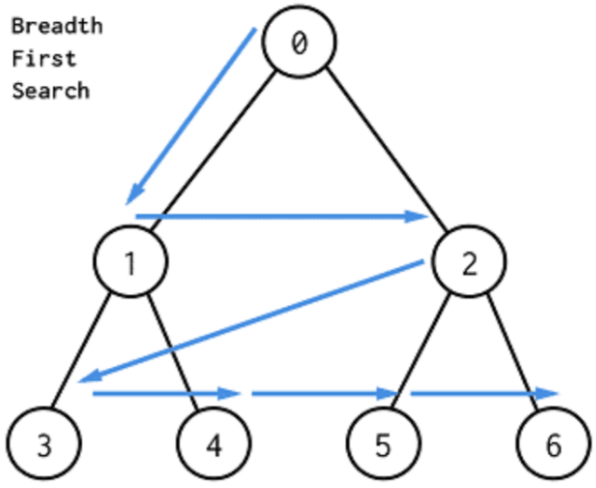
2.1.1. Graph Traversal

Graph traversal adalah proses penelusuran simpul-simpul dalam graf secara sistematis. Proses penelusuran graf dengan algoritma ini dilakukan sesuai aturan dan ketentuan tertentu. Terdapat 2 jenis algoritma graf transversal yaitu *breadth first search (BFS)* dan *depth first search (DFS)*.

2.1.2. Algoritma BFS

Breadth First Search (BFS) adalah metode penelusuran graf secara traversal secara melebar. Pencarian dimulai dari sebuah simpul v yang selanjutnya akan "diperlebar" ke simpul-simpul yang bertetanggaan dengan simpul v. Langkah-langkah yang dilakukan pada *BFS* adalah sebagai berikut:

a. Kunjungi simpul v
b. Kunjungi semua simpul yang bertetangga dengan simpul v terlebih dahulu
c. Untuk setiap simpul yang dikunjungi tersebut kunjungi simpul yang belum dikunjungi dan bertetangga simpul tersebut
d. Ulangi langkah 3 hingga semua simpul dikunjungi

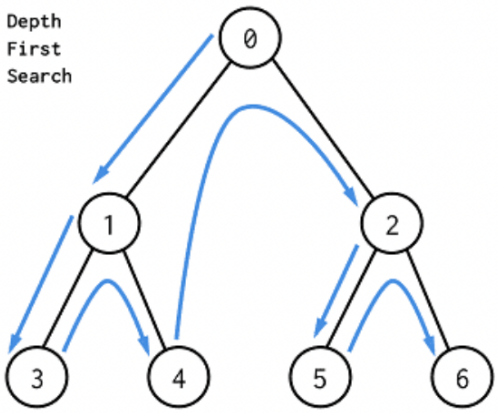


Gambar 2.1.2.1 Ilustrasi Algoritma BFS

2.1.3. Algoritma DFS

Depth First Search (DFS) adalah metode penelusuran graf secara traversal secara mendalam. Penelusuran akan dimulai dari sebuah simpul v yang selanjutnya akan diekspansi secara "mendalam", sehingga penelusuran akan dilakukan ke simpul-simpul daun terlebih dahulu, jika sudah tidak ada simpul yang dapat dikunjungi maka akan dilakukan proses *backtracking,* hal ini bertujuan untuk mencari simpul selanjutnya yang belum dikunjungi. Berikut langkah-langkah dari algoritma DFS:

a. Kunjungi simpul v
b. Kunjungi simpul w yang merupakan simpul tetangga dari v
c. Ulangi algoritma DFS secara rekursif dengan simpul awal adalah simpul w
d. Apabila proses pencarian mencapai suatu simpul u sehingga tidak ada lagi simpul tetangga yang belum dikunjungi, dilakukan pencarian runut-balik ke simpul terakhir yang dikunjungi sebelum simpul u dan memiliki simpul tetangga yang belum dikunjungi.
e. Pencarian berakhir apabila semua simpul telah dikunjungi atau tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai.



Gambar 2.1.3.1 Ilustrasi Algoritma DFS

2.2. C# Desktop Application Development

Desktop application merupakan perangkat lunak yang berjalan secara lokal pada komputer pribadi atau laptop pengguna, berbeda dengan *web application* yang berjalan pada *web browser* maupun *mobile application* yang berjalan pada *smartphone*. Desktop application dapat hanya berbentuk CLI (*Command Line Interface*) atau memiliki GUI (*Graphical User Interface*).

Salah satu bahasa yang populer digunakan dalam pengembangan aplikasi *desktop* adalah C#. C# merupakan bahasa pemrograman yang dikembangkan oleh *Microsoft* dan berjalan pada *framework* .NET. Bahasa C# tergolong bahasa pemrograman berorientasi objek. Dalam proses pengembangan aplikasi *desktop* berbahasa C#, *programmer* biasa menggunakan sebuah IDE buatan *Microsoft* yang bernama *Visual Studio* untuk mempermudah proses pengembangan.

BAB III
ANALISIS PEMECAHAN MASALAH

3.1. Langkah-Langkah Pemecahan Masalah

Untuk menyelesaikan masalah yang sudah disebutkan di atas, diperlukan beberapa langkah-langkah sebagai berikut:

1. Memahami permasalahan yang diberikan
2. Memahami konsep algoritma BFS dan DFS dalam traversal graf
3. Mempelajari bahasa C#
4. Memetakan persoalan menjadi elemen-elemen BFS dan DFS
5. Menggambarkan *input* file .txt menjadi simpul dan sisi yang membentuk graf
6. Mengimplementasikan algoritma BFS dan DFS dalam bahasa C# untuk menyelesaikan persoalan
7. Membuat visualisasi graf menggunakan MSAGL
8. Membuat GUI
9. Menghubungkan GUI dengan program utama

Di samping itu, langkah-langkah untuk mencari *file* spesifik dengan memanfaatkan algoritma *Depth First Search* dan *Breadth First Search* adalah sebagai berikut.

1. Program akan menerima *input* melalui *graphical user interface*. *Input* berupa *directory root*, nama *file* yang akan dicari, waktu *delay* tiap langkah, metode cari dan *check* "Find All Occurrence".
2. Program akan menjalankan *method* `searchFilePathDFS` jika metode cari yang dipilih adalah DFS. Sedangkan, jika metode cari yang dipilih adalah BFS maka program akan menjalankan *method* `searchFilePathBFS`.
    2.1. *Method* `searchFilePathDFS` akan mencari file secara DFS yaitu dengan mengunjungi simpul tetangga pertama secara terus menerus hingga file ditemukan atau simpul tidak memiliki tetangga. Maka, akan runut balik ke simpul sebelumnya dan mengunjungi simpul selanjutnya. Setiap mengunjungi simpul graph akan menambahkan simpul dan sisi baru. Program akan berhenti ketika file ditemukan. Namun, jika *Find All*

Occurrence diaktifkan maka program akan lanjut hingga seluruh *subdirectories* sudah dikunjungi. Jika file tidak ada dalam maka program akan berhenti ketika seluruh *subdirectories* sudah dikunjungi.

2.2. Method `searchFilePathBFS` akan mencari file secara BFS yaitu dengan mengunjungi semua simpul tetangga. Setiap simpul yang dikunjungi tersebut kunjungi simpul yang belum dikunjungi dan bertetangga simpul tersebut. Setiap mengunjungi simpul graph akan menambahkan simpul dan sisi baru. Program akan berhenti ketika file ditemukan. Namun, jika *Find All Occurrence* diaktifkan maka program akan lanjut hingga seluruh *subdirectories* sudah dikunjungi. Jika file tidak ada dalam maka program akan berhenti ketika seluruh *subdirectories* sudah dikunjungi.

3.2. Proses *Mapping* Persoalan menjadi Elemen Algoritma BFS dan DFS

3.2.1. Breadth First Search

Algoritma BFS akan dimulai dari sebuah simpul *root* sebagai simpul *parent* saat ini berupa *directory* awal dimana proses pencarian akan dimulai. Simpul *parent* akan dimasukan kedalam sebuah *queue* dan *graph*. Simpul tetangganya akan dimasukan ke dalam *queue* dan *graph*. Simpul *parent* akan dikeluarkan dari *queue*. Simpul selanjutnya yang ada di dalam *queue* akan menjadi simpul *parent* saat ini yang baru. Program akan terus berjalan hingga *file* ditemukan atau *queue* habis. *Directory* dari *root* sampai *file* akan dicatat.

3.2.2. Depth First Search

Algoritma DFS akan dimulai dari sebuah simpul *root* berupa *directory* awal dimana proses pencarian akan dimulai. Simpul *root* akan dimasukan ke dalam sebuah *graph*. Program akan mengunjungi simpul tetangga pertama secara terus menerus hingga *file* ditemukan atau simpul tidak memiliki tetangga. Maka, akan runut balik ke simpul sebelumnya dan mengunjungi simpul selanjutnya. Program akan terus berjalan hingga *file* ditemukan atau *queue* habis. *Directory* dari *root* sampai *file* akan dicatat.

3.3. Ilustrasi Kasus Lain

3.3.1. Kasus 1 Pencarian *File* dari Substring Nama *File*

Berbeda dengan spesifikasi tugas besar ini, *file* yang dicari tidak harus menginput nama *file* yang eksak. File yang dicari adalah *file* yang substringnya sama dengan *input* nama *file* yang dicari. Ini akan membuat pencarian *file* yang lebih luas. *File* dapat dicari melalui kata kunci dari nama *file* sehingga tidak perlu mengetahui nama *file* secara eksak.

3.3.2. Kasus 2 Pencarian File dari Kata-Kata yang Mirip

Pada kasus ini, *input filename* yang dimasukan dapat terdiri dari beberapa kata yang menjadi kata kunci *file* yang akan dicari. *File* akan ditentukan relevansinya dengan menggunakan algoritma ruang vektor. *File-file* yang paling relevan akan dimunculkan di paling atas.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1. Implementasi Program

Pada program kami, hasil implementasi algoritma BFS dan DFS terdapat pada *file* BFS.cs dan DFS.cs. Di dalam *file* tersebut terdapat sebuah fungsi pemanggilan yang utama, yakni main. Secara keseluruhan, berikut ini merupakan implementasi kode dari program *folder crawler* kami yang ditulis dalam *pseudocode.*

4.1.1. Static Void main

```
procedure main()
{Program utama untuk mencari file spesifik dari sebuah root directory berdasarkan strategi
DFS atau BFS}

KAMUS LOKAL
    start, search, rootDir, filename, startingDirectory : string
    startingNode : Node
    stepDelay : integer
    FindAll : boolean
    gViewer : GViewer

ALGORITMA
    if (searchMethod = "DFS") then
        dfs ← new DFS(startingDirectory, gViewer)
        dfs.searchFilePathDFS(startingDirectory, filename, null, stepDelay, findAllOccurance)
    else if (searchMethod = "BFS") then
        bfs ← new BFS(startingDirectory, gViewer)
        bfs.searchFilePathBFS(startingDirectory, filename, stepDelay, findAllOccurance)
```

4.1.2. Public Void searchFilePathDFS

```
procedure searchFilePathDFS(string start, string search, Node startingNode, integer
        stepDelay, boolean findAll)
{mencari file spesifik dari sebuah root directory berdasarkan strategi DFS}

KAMUS LOKAL
    root : System.IO.DirectoryInfo
    files : array of System.IO.FileInfo
    subDirs : array of System.IO.DirectoryInfo
    startingNode, subdirectory, N, nextNode : Node
    i : integer
ALGORITMA
    if (solutionPath.Count > 0 and not findAll) then
        →
    {Inisialisasi}
    root ← new System.IO.DirectoryInfo(start)
    files ← null
    subDirs ← null
```

```
    if (startingNode = null) then
        startingNode ← fileGraph.R
    files ← root.GetFiles("*.*")
    if (files ≠ null) then
        {Ubah node menjadi warna merah jika warna awalnya bukan}
        if (startingNode.Attr.Color ≠ Microsoft.Msagl.Drawing.Color.Blue) then
            startingNode.Attr.Color ← Microsoft.Msagl.Drawing.Color.Red

        {Tambahkan node semua folder start directory}
        subDirs ← root.GetDirectories()

        {Dilakukan Reverse agar pembentukan pohon rapih}
        Array.Reverse(subDirs)

        for i←0 to subDirs.length() do
            subdirectory ← fileGraph.AddEdgeBlack(startingNode, subDirs[i].Name)
            Array.Reverse(subDirs)

        {Tambahkan node semua file start directory}
        Array.Reverse(files)
        for i←0 to files.length() do
            subdirectory ← fileGraph.AddEdgeBlack(startingNode, files[i].Name)
        Array.Reverse(files)

        {Tampilkan pohon}
        fileGraph.showGraph(viewer, stepDelay)

        for i←0 to files.length() do
            {Mengecek apakah file merupakan yang dicari}
            if (search.Equals(files[i].Name)) then
                this.addSolution(files[i].FullName)
                fileGraph.TurnBlue(fileGraph.dirToList(files[i].FullName))

                {Tampilkan pohon}
                fileGraph.showGraph(viewer, stepDelay)

                if (not findAll) then
                    break
            else
                N ← fileGraph.ColorEdgeRed(startingNode, files[i].Name)
                FileGraph.ColorNodeRed(N)

                {Tampilkan pohon}
                fileGraph.showGraph(viewer, stepDelay)

        {Melanjutkan DFS}
        if (findAll or this.solutionPath.Count = 0) then
            for i←0 to subDirs.length() do
                nextNode ← fileGraph.ColorEdgeRed(startingNode, subDirs[i].Name)
                searchFilePathDFS(subDirs[i].FullName, search, nextNode, stepDelay, findAll)
```

4.1.3. Public Void searchFilePathBFS

```
procedure searchFilePathBFS(string rootDir, string filename, integer stepDelay, boolean
FindAll)
{mencari file spesifik dari sebuah root directory berdasarkan strategi BFS}

KAMUS LOKAL
```

```
    res : List of string
    q : Queue of string
    parentNodeQueue : Queue of Node
    dir, dirName : string
    ParentNode, currentParentNode, N : Node
    directories, files : array of string
    i : int
ALGORITMA
    res ← new List<string>()
    q ← new Queue<string>()
    parentNodeQueue ← new Queue<Node>()
    dir ← rootDir
    {Add node di awal untuk root}
    q.Enqueue(rootDir)
    parentNodeQueue.Enqueue(fileGraph.R)

    ParentNode ← fileGraph.R
    ParentNode.Attr.Color ← Microsoft.Msagl.Drawing.Color.Red

    {Implementasi}
    while (q.Count > 0 and (FindAll or this.solutionPath.Count = 0)) do
        {Warnai node yang sedang dicek menjadi merah}
        dir ← q.Dequeue()
        ParentNode ← parentNodeQueue.Dequeue()

        currentParentNode ← fileGraph.R

        if (dir ≠ rootDir) then
            currentParentNode ← fileGraph.ColorEdgeRed(ParentNode, Path.GetFileName(dir))
            currentParentNode.Attr.Color ← Microsoft.Msagl.Drawing.Color.Red

        fileGraph.showGraph(this.viewer, stepDelay)

        directories ← Directory.GetDirectories(dir, "*")

        {Tambahkan node semua folder directory}
        Array.Reverse(directories)

        for i←0 to directories.length() do
            N ← fileGraph.AddEdgeBlack(currentParentNode, Path.GetFileName(directories[i]))

        Array.Reverse(directories)

        {Tambahkan node semua file directory}
        files ← Directory.GetFiles(dir)

        Array.Reverse(files)

        for i←0 to files.length() do
            N ← fileGraph.AddEdgeBlack(currentParentNode, Path.GetFileName(files[i]))

        Array.Reverse(files)

        {Tampilkan pohon}
        fileGraph.showGraph(this.viewer, stepDelay)

        {Search file}
        for i←0 to files.length() do
            if (Path.GetFileName(files[i]).Equals(filename)) then
                this.setSolution(files[i])
```

```
            fileGraph.TurnBlue(fileGraph.dirToList(files[i]))
            fileGraph.showGraph(this.viewer, stepDelay)
            if (not FindAll) then
                break
        else
            N ← fileGraph.ColorEdgeRed(currentParentNode, Path.GetFileName(files[i]))
            FileGraph.ColorNodeRed(N)
            fileGraph.showGraph(this.viewer, stepDelay)

    {Search directory}
    if (FindAll or this.solutionPath.Count = 0) then
        for i←0 to directories.length() do
            dirName ← Path.GetFileName(directories[i])

            q.Enqueue(directories[i])
            parentNodeQueue.Enqueue(currentParentNode)
```

4.2. Struktur Data dan Spesifikasi Program

Program folder crawler ini diimplementasikan dengan bahasa pemrograman C# dan dikembangkan dengan IDE Visual Studio. Visualisasi pohon pencarian *file* diimplementasikan menggunakan kakas MSAGL dan .NET. Struktur data yang digunakan berbasis pada kelas. Di dalam program ini, telah terdefinisi beberapa kelas, antara lain *BFS, DFS, FileGraph, Form1,* dan *Main*. Kelas-kelas tersebut kemudian kami kembangkan dan lengkapi dengan penjelasan sebagai berikut.

4.2.1. BFS

Kelas ini berisi strategi cara mencari *file* spesifik dari sebuah *root directory* berdasarkan algoritma BFS. Dalam pencarian menggunakan metode BFS, digunakan struktur data *list of string* untuk mencatat solusi-solusi pencarian. Selain itu, digunakan juga struktur data *queue of string* untuk mencatat *directory* yang akan ditelusuri dan *queue of node* untuk mencatat *parent* dari *node* yang akan ditelusuri.

4.2.2. DFS

Kelas ini berisi strategi cara mencari *file* spesifik dari sebuah *root directory* berdasarkan algoritma DFS. Dalam pencarian menggunakan metode BFS, digunakan struktur data *list of string* untuk mencatat solusi-solusi pencarian.

4.2.3. FileGraph

Kelas ini berisi hasil implementasi kakas MSAGL dalam menampilkan visualisasi pohon pencarian *file* berdasarkan informasi direktori dari *folder* yang di-*input*.

4.2.4. Form1

Kelas ini berisi hasil implementasi *graphical user interface* program. GUI digunakan dalam input *root directory*, f*ilename* yang akan dicari, check *Find All Occurrence*, waktu *delay* langkah, dan metode pencarian.

4.2.5. Main

Kelas ini berfungsi untuk memanggil kelas-kelas lain dan menjalankan program dengan menjalankan seluruh algoritma yang telah dikembangkan.

4.3. Tata Cara Penggunaan Program

Gambar 4.3.1 Tampilan Program

1. Pilih *directory* dengan menekan tombol “Select Directory”[1] untuk memilih *root directory*

2. Masukan *file* yang akan dicari dalam kolom “File Name”[2]
3. Cek “Find All Occurrence”[3] untuk mencari semua *file* sesuai dengan “File Name”[2], jika tidak di cek maka hanya akan mencari satu *file* yang pertama kali sesuai.
4. Masukan “Step Delay”[4] yaitu interval waktu antar langkah pencarian
5. Pilih metode pencarian “DFS”[5] atau “BFS”[6]
6. Tekan tombol “Search”[7] untuk memulai pencarian

Fitur :

[1] Tombol Select Directory

[2] kolom untuk memasukan File Name yang akan dicari

[3] Check untuk Find All Occurance

[4] Kolom untuk memasukan Step Delay

[5] Tombol DFS

[6] Tombol BFS

[7] Tombol Search

[8] Tombol Home

[9] Tombol Zoom In

[10] Tombol Zoom Out

[11] Tombol Zoom In by dragging a rectangle

[12] Tombol Pan

[13] Tombol Backward

[14] Tombol Forward

[15] Tombol Save Graph

[16] Tombol Undo

[17] Tombol Redo

[18] Tombol Load Graph

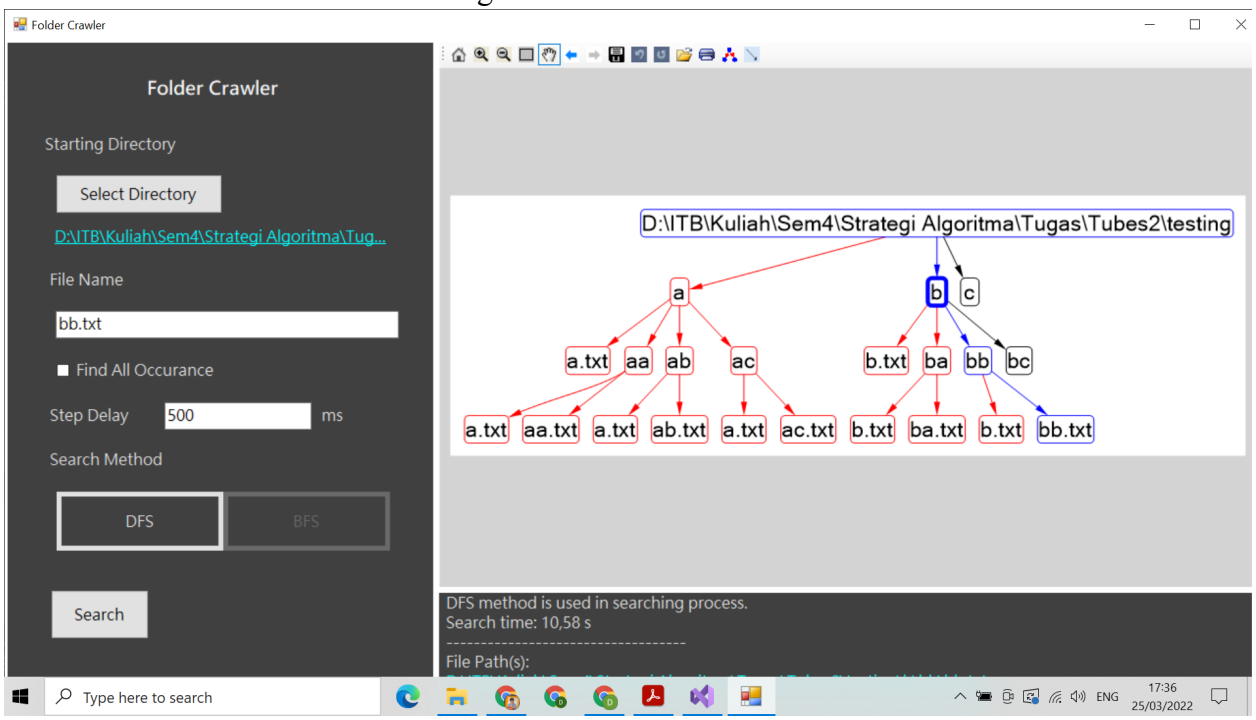
[19] Tombol Print

[20] Tombol Configure Layout Settings

[21] Tombol Edge Insertion

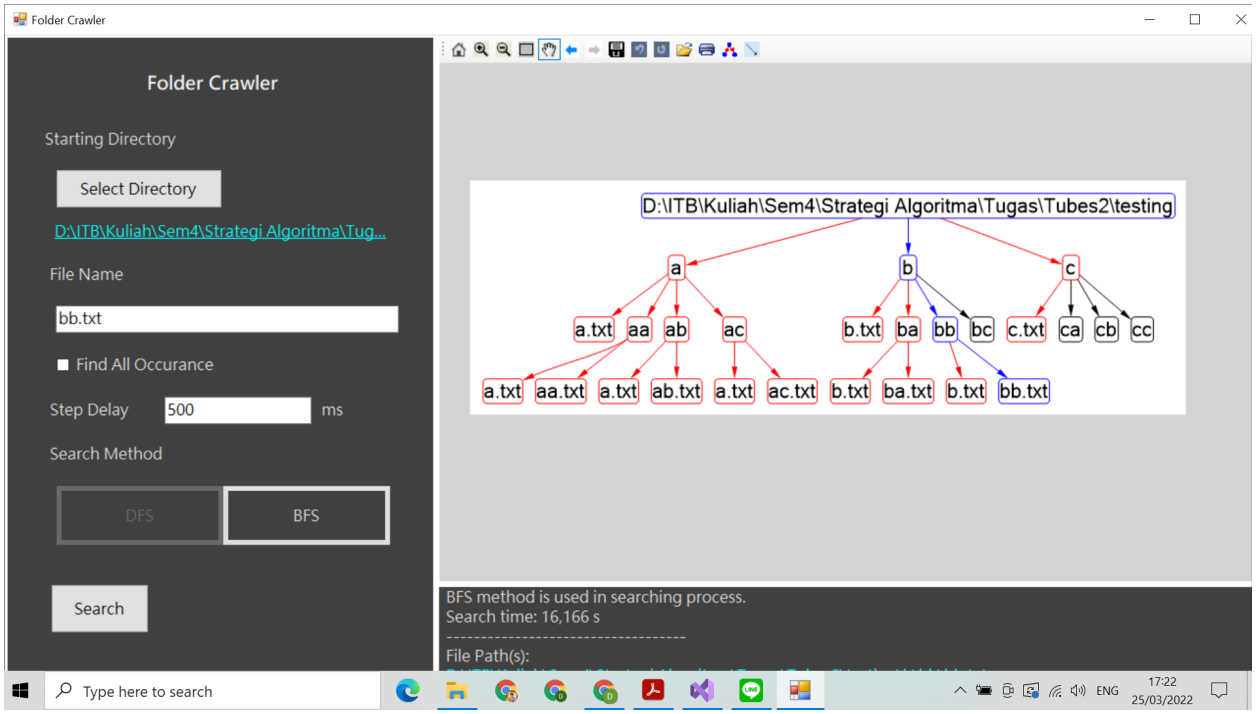
4.4. Hasil Pengujian

4.4.1. Mencari Satu *File* dengan Metode DFS



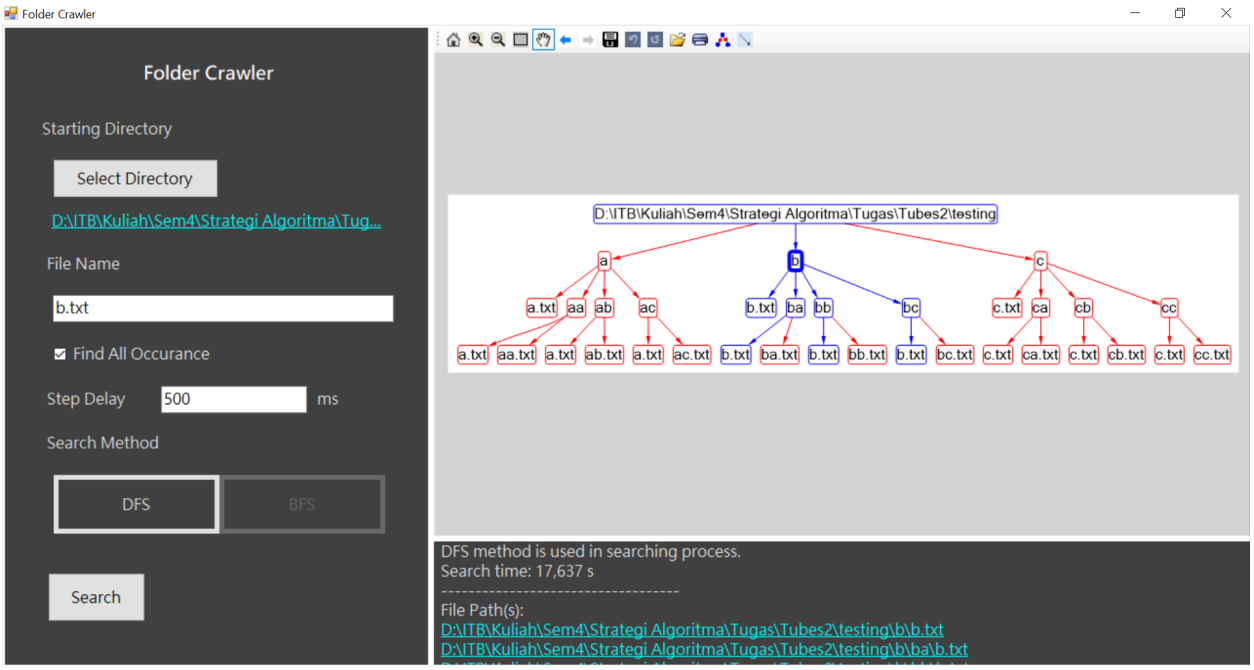
Gambar 4.4.1.1 Hasil *Testing* Mencari Satu *File* dengan Metode DFS

4.4.2. Mencari Satu *File* dengan Metode BFS



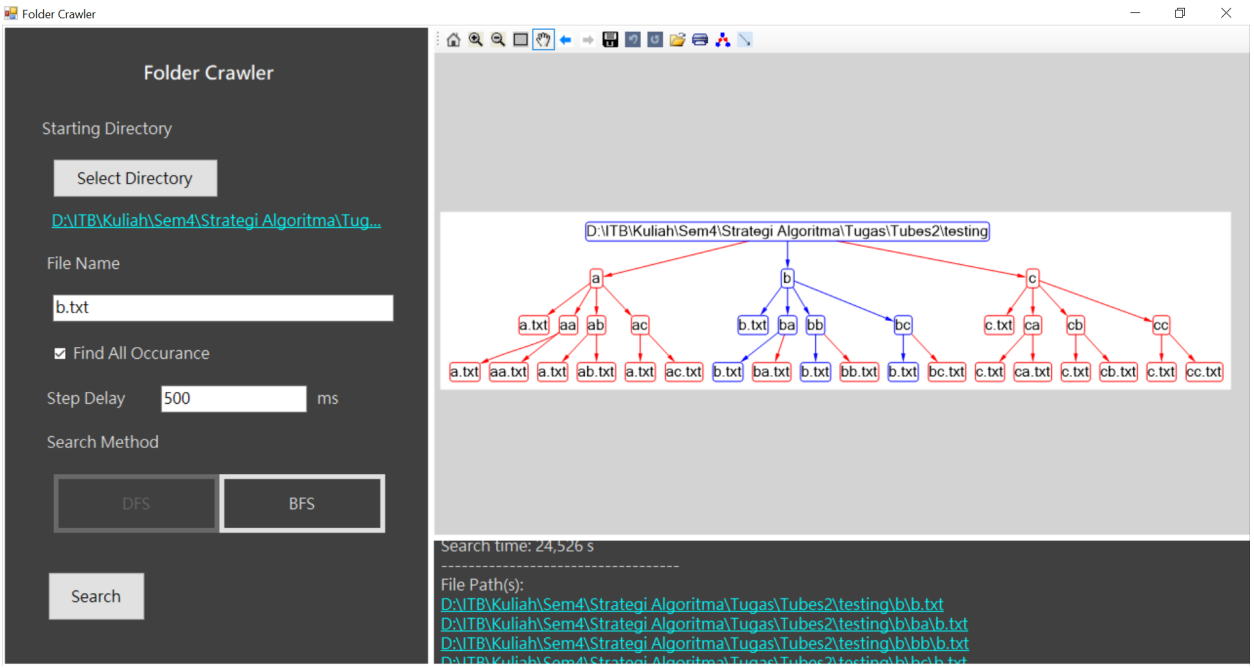
Gambar 4.4.2.1 Hasil *Testing* Mencari Satu *File* dengan Metode BFS

4.4.3. Mencari Semua *File* dengan Metode DFS



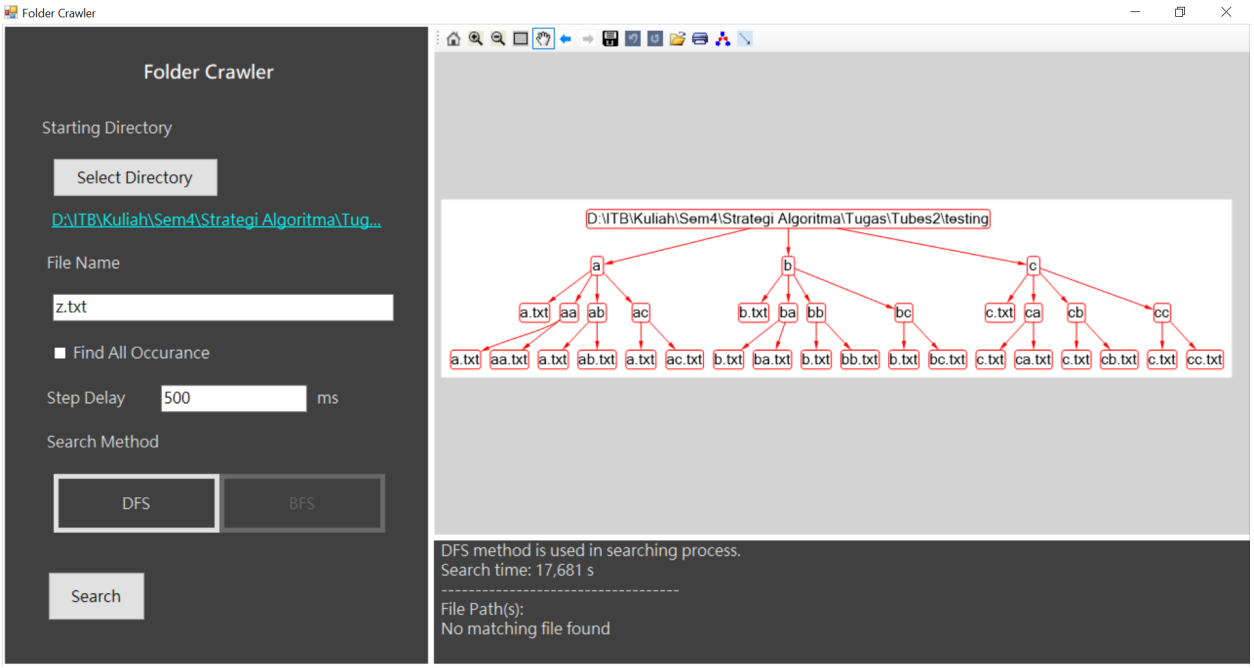
Gambar 4.4.3.1 Hasil *Testing* Mencari Semua *File* dengan Metode DFS

4.4.4. Mencari Semua *File* dengan Metode BFS



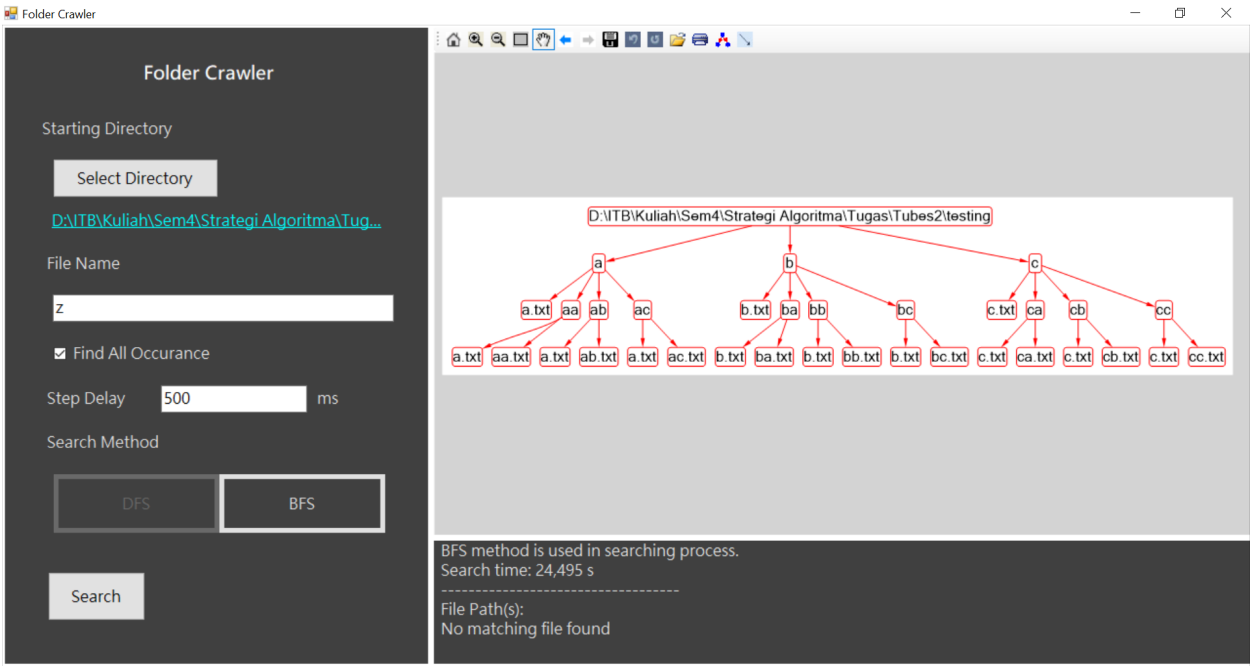
Gambar 4.4.4.1 Hasil *Testing* Mencari Semua *File* dengan Metode BFS

4.4.5. Mencari *File* yang Tidak Ada dengan Metode DFS



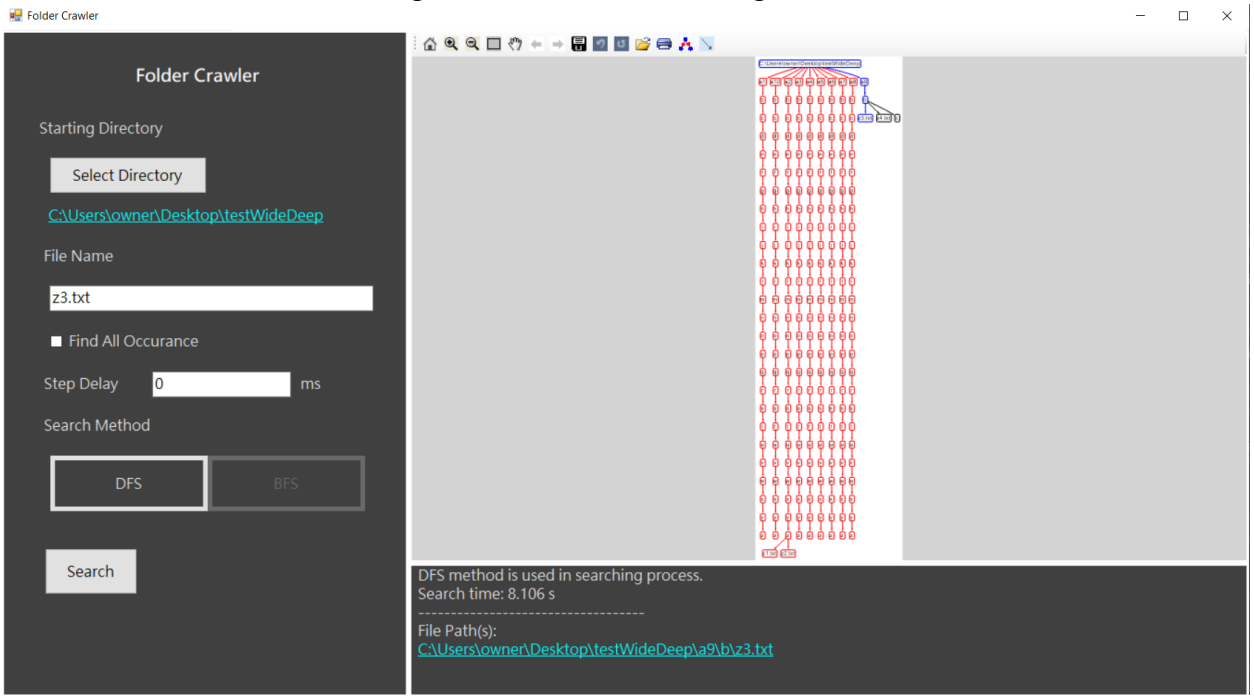
Gambar 4.4.5.1 Hasil *Testing* Mencari *File* yang Tidak Ada dengan Metode DFS

4.4.6. Mencari *File* yang Tidak Ada dengan Metode BFS



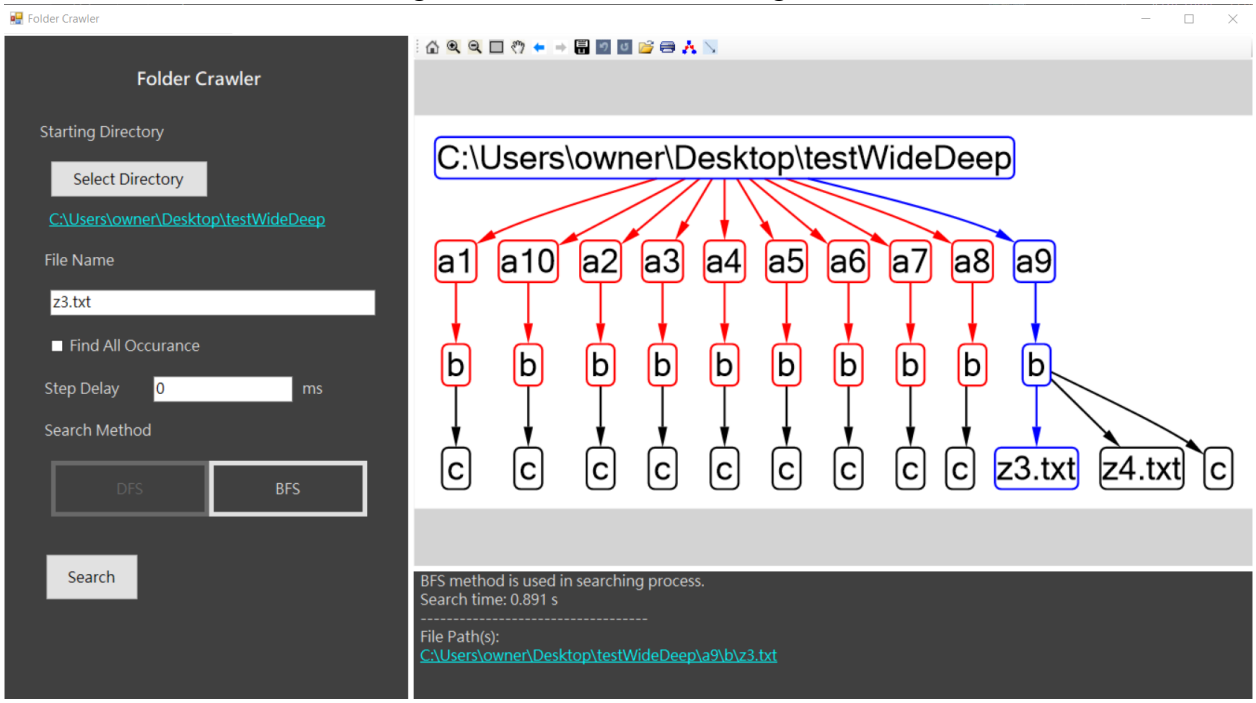
Gambar 4.4.6.1 Hasil *Testing* Mencari *File* yang Tidak Ada dengan Metode BFS

4.4.7. Mencari *File* dengan kedalaman rendah dengan Metode DFS



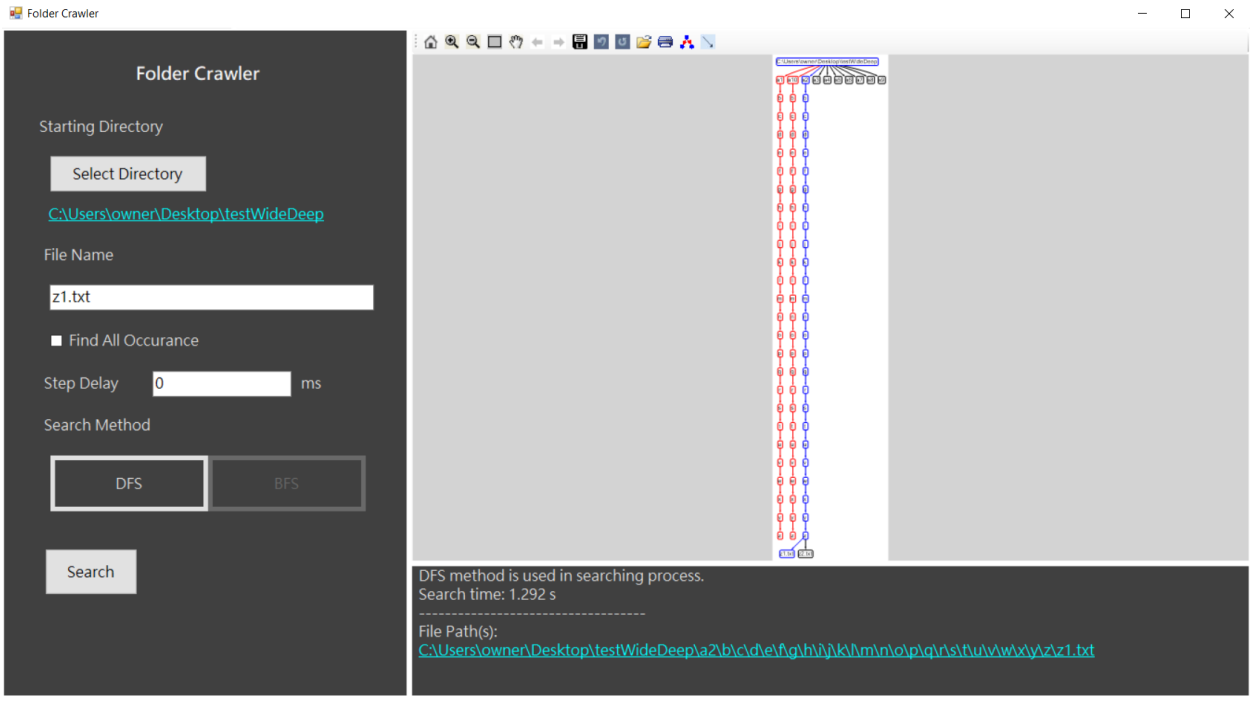
Gambar 4.4.7.1 Hasil *Testing* Mencari *File* dengan Kedalaman Rendah dengan Metode DFS

4.4.8. Mencari *File* dengan Kedalaman Rendah dengan Metode BFS



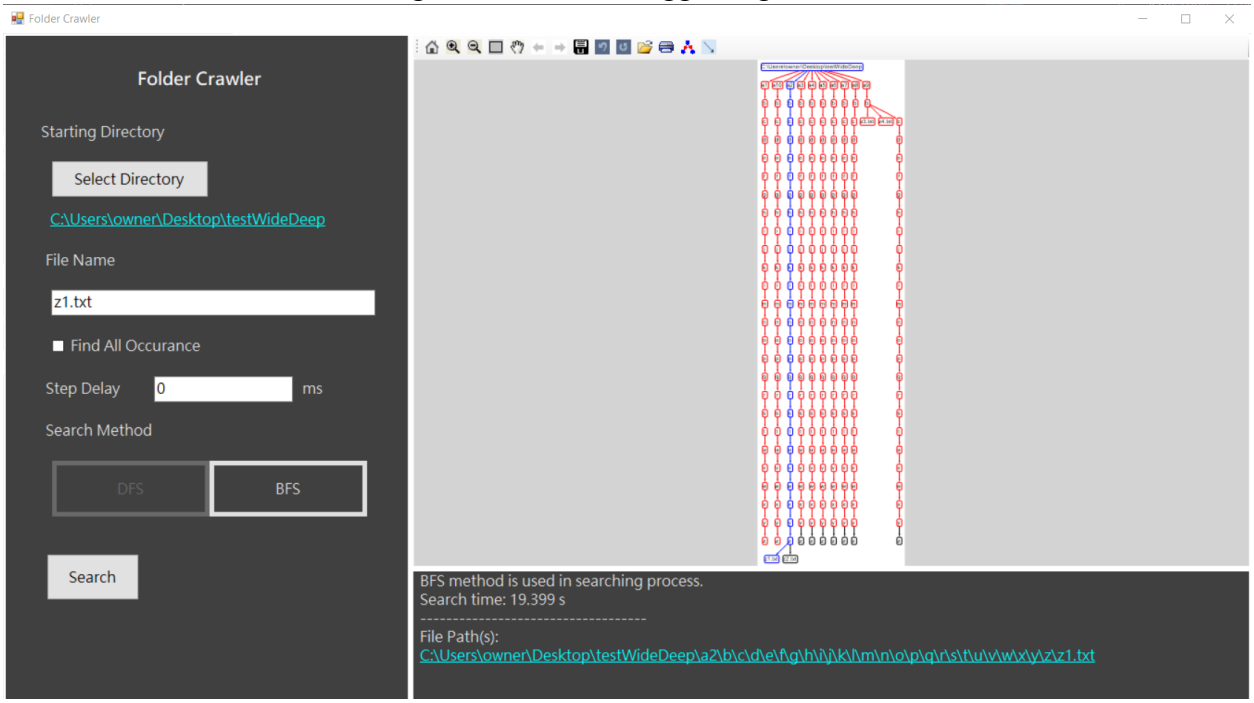
Gambar 4.4.8.1 Hasil *Testing* Mencari *File* dengan Kedalaman Rendah dengan Metode BFS

4.4.9. Mencari *File* dengan Kedalaman Tinggi dengan Metode DFS



Gambar 4.4.9.1 Hasil *Testing* Mencari *File* dengan Kedalaman Tinggi dengan Metode DFS

4.4.10. Mencari *File* dengan Kedalaman Tinggi dengan Metode BFS



Gambar 4.4.10.1 Hasil *Testing* Mencari *File* dengan kedalaman tinggi dengan Metode BFS

4.5. Analisis Desain Solusi Algoritma BFS dan DFS

Berdasarkan hasil pengujian program yang kami buat, pencarian *file* dengan algoritma DFS cenderung lebih cepat eksekusinya dibandingkan dengan algoritma BFS. Misalnya, pada gambar 4.4.3.1 pencarian dengan DFS membutuhkan waktu 17,637 detik sedangkan pada gambar 4.4.4.1 pencarian *file* dengan konfigurasi yang sama menggunakan BFS membutuhkan waktu 24,526 detik. Perbedaan durasi tersebut terjadi karena BFS membutuhkan pemrosesan yang lebih rumit baik pada proses pencarian maupun penunjukan pohon seperti penggunaan *queue* dan penyimpanan *node*.

Walaupun begitu, DFS dan BFS menunjukkan kelebihannya masing-masing pada kasus yang ekstrim. Contohnya, jika *file* yang dicari berada pada kedalaman yang rendah dan terdapat banyak *folder* di *directory*, BFS akan memerlukan waktu pencarian yang jauh lebih rendah dibandingkan DFS. Bisa dilihat pada gambar 4.4.7.1 pencarian *file* dengan DFS memerlukan waktu 8,106 detik sedangkan pada gambar 4.4.8.1 pencarian serupa dengan BFS memerlukan hanya 0,891 detik.

Sebaliknya, jika *file* berada di kedalaman yang tinggi dan berada di *folder* yang pencariannya lebih awal, pencarian dengan metode DFS lebih unggul. Pada gambar 4.4.9.1 pencarian *file* menggunakan DFS memerlukan waktu 1,292 detik sedangkan pada gambar 4.4.10.1 pencarian serupa dengan BFS memerlukan waktu yang jauh lebih lama, yaitu 19,399 detik.

BAB V

KESIMPULAN DAN SARAN

5.1. Kesimpulan

Telah berhasil diimplementasikan sebuah program berupa *folder crawler* yang dirancang dan dikembangkan untuk melakukan pencarian terhadap suatu file tertentu dalam suatu *directory* sesuai dengan yang diminta dalam spesifikasi Tugas Besar 2 IF2211 Strategi Algoritma Semester 2 Tahun 2021/2022. Hal mengenai program *folder crawler* yang berhasil diimplementasikan dalam program ini meliputi:

1. Konsep algoritma BFS dan DFS, serta penerapannya dalam implementasi program *folder crawler*,
2. Visualisasi hasil dari pencarian *file/folder* dalam bentuk pohon,
3. Analisis desain solusi algoritma BFS dan DFS, serta perbandingan antar tingkat efektivitas program dari masing-masing algoritma,
4. Implementasi penggunaan pustaka atau kakas untuk melakukan visualisasi, yakni MSAGL,
5. Implementasi program dengan paradigma pemrograman berorientasi objek untuk menciptakan *folder crawle*r yang dapat memodelkan fitur dari *file explorer* dengan menggunakan framework .NET.

Semua implementasi dari konsep-konsep di atas kemudian berhasil digunakan untuk menyelesaikan seluruh fitur yang ada di dalam spesifikasi. Fitur-fitur tersebut telah terdapat pada program *folder crawler* yang kami buat. Setidaknya terdapat 6 fitur utama yang dapat digunakan pada *folder crawler* kami, antara lain:

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* 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. 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.

Fitur ini dapat digunakan pada *folder crawler* kami yang sudah diciptakan sedemikian rupa dengan memanfaatkan tampilan GUI yang *user-friendly* dan menarik untuk dapat mempermudah interaksi antara *user* dengan aplikasi.

Dengan pengimplementasian berbagai algoritma pencarian *file* sesuai dengan *query*, dimulai dari *starting directory* hingga seluruh *children* dari starting *directory*, khususnya dengan menggunakan konsep algoritma BFS dan DFS, kita dapat memecahkan permasalahan pencarian *file* yang dapat kita temukan dalam kehidupan sehari-hari. Konsep yang telah diajarkan di perkuliahan IF2211 dapat dengan baik diterapkan dalam pengerjaan Tugas Besar 2 IF2211 Strategi Algoritma ini. Selain melakukan eksplorasi terhadap berbagai alternatif algoritma pencarian *file* yang dapat dipakai, kelompok juga berhasil memvisualisasikan pohon hasil pencarian *file* dengan melakukan eksplorasi terhadap kakas MSAGL yang dianggap dapat membantu proses pengembangan hasil visualisasi.

Dengan demikian, kelompok menyimpulkan bahwa dengan mengerjakan Tugas Besar 2 IF2211 Strategi Algoritma Semester 2 Tahun 2021/2022 ini, dapat diketahui bahwa untuk menyelesaikan suatu masalah yang mungkin ditemukan dalam kehidupan sehari-hari, dalam hal ini misalnya, melakukan pencarian *file* tertentu yang sudah tidak kita ingat pasti letak penyimpanannya, dapat diimplementasikan program *folder crawler* yang memodelkan fitur dari *file explorer* untuk melakukan pencarian *file* berdasarkan input *query* yang diinginkan sebagai bentuk penerapan dari konsep algoritma BFS dan DFS yang telah dipelajari pada kuliah IF2211.

5.2. Saran

Tugas Besar 2 IF2211 Strategi Algoritma Semester 2 Tahun 2021/2022 menjadi salah satu proses pembelajaran bagi kelompok dalam menerapkan ilmu-ilmu yang diajarkan pada kuliah maupun melakukan eksplorasi materi secara mandiri. Berikut ini merupakan sejumlah saran dari kelompok untuk pihak-pihak yang ingin melakukan atau mengerjakan hal serupa.

1. Program yang diminta adalah program dengan menggunakan bahasa pemrograman C#, yakni salah satu bahasa pemrograman yang belum dikuasai secara menyeluruh oleh ketiga anggota kelompok yang terlibat dalam pengerjaan tugas besar ini. Dengan demikian, kelompok merekomendasikan agar disediakan waktu yang cukup untuk melakukan eksplorasi terkait bahasa pemrograman yang digunakan sebelum mengimplementasikannya ke dalam sebuah program. Hal ini akan meningkatkan efektivitas kerja tim dalam pembuatan suatu program. Di samping itu, perlu dipertimbangkan pula waktu yang dimiliki untuk melakukan eksplorasi terhadap suatu bahasa pemrograman atau *framework* tertentu sehingga tidak membebani *programmer* dalam pengerjaan proyek dengan jangka waktu yang singkat. Gunakan kemampuan berpikir serta bekerja yang elaboratif dan koordinatif di antara seluruh anggota kelompok yang terlibat.

2. Modularitas menjadi hal yang krusial dalam menciptakan suatu program secara efektif dan efisien. Dalam jangka waktu yang singkat, pemrograman secara modular dapat membantu *programmer* untuk memudahkan proses pencarian kesalahan/*error* serta *debugging*. Pada dasarnya, memrogram secara modular berarti memecah-mecah program menjadi modul-modul kecil di mana masing-masing modul berinteraksi melalui antarmuka modul. Masalah yang awalnya kompleks dapat dibagi menjadi bagian-bagian kecil yang lebih sederhana dan dapat diselesaikan dalam lingkup yang lebih kecil. Akibatnya, apabila terdapat *error/bug* pada program, kesalahan dapat dengan mudah ditemukan karena alur logika yang jelas serta dapat dilokalisasi dalam satu modul. Lebih dari pada itu, modifikasi program dapat dilakukan tanpa mengganggu *body* program secara keseluruhan. Oleh karena itu, kelompok sangat menyarankan untuk melakukan pemrograman secara modular dalam mengimplementasikan algoritma BFS dan DFS dalam pembuatan program *folder crawling* ini.
3. Penting bagi kelompok untuk memiliki strategi serta distribusi tugas yang baik. Ketika membuat program dalam sebuah tim, kesamaan cara menulis kode serta kemampuan untuk menulis komentar menjadi hal yang sangat penting. Hal ini diperlukan agar memudahkan anggota kelompok dalam menyatukan dan melanjutkan sebuah program. Kemampuan tersebut tentunya didukung juga dengan adanya *version control system* yang baik yang dapat digunakan oleh *programmer* dalam membuat sebuah program secara bersama-sama. Untuk itu, kami sangat menyarankan 'GitHub' untuk digunakan sebagai *version control system* dalam pengerjaan tugas-tugas besar pada mata kuliah IF2211 ini, maupun pada pembuatan program dan pengerjaan proyek yang lainnya.
4. Kelompok menyadari bahwa pada implementasi program *folder crawler* yang telah kami buat, masih banyak aspek yang dapat dikembangkan lebih lagi. Salah satunya ialah dengan mengoptimalkan algoritma BFS dan DFS yang digunakan agar proses pencarian *file* pada *directory* yang dituju dapat berlangsung lebih cepat. Program juga dapat dikembangkan dari sisi UI/UX yang telah diimplementasikan oleh kelompok dalam bentuk GUI. Hal ini tentu menjadi ruang untuk *programmer* agar dapat melakukan improvisasi terhadap implementasi dan pengembangan program *folder crawler*, terutama dalam hal eksplorasi algoritma dan desain UI/UX. Selain itu, berhubung program ini merupakan pengembangan aplikasi *desktop* berbahasa C#, kelompok juga merekomendasikan untuk menggunakan sebuah IDE buatan *Microsoft* yang bernama *Visual Studio* untuk mempermudah proses pengembangan.

LAMPIRAN

Link *repository* GitHub:

https://github.com/nelsenputra/Tubes2_13520126

DAFTAR PUSTAKA

Munir, Rinaldi. (2020). Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1). Institut Teknologi Bandung. http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf. Diakses pada 15 Maret 2022.

Munir, Rinaldi. (2020). Breadth First Search (BFS) dan Depth First Search (DFS)(Bagian 2). Institut Teknologi Bandung. http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf. Diakses pada 15 Maret 2022.