

LAPORAN TUGAS BESAR

Pengaplikasian Algoritma BFS dan DFS dalam Implementasi *Folder Crawling*

Ditujukan untuk memenuhi salah satu tugas besar mata kuliah IF2211 Strategi Algoritma
pada Semester II Tahun Akademik 2021/2022

Disusun oleh:

Grace Claudia (K3)	13520078
Sarah Azka Arief (K2)	13520083
Rania Dwi Fadhilah (K1)	13520142



PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2022

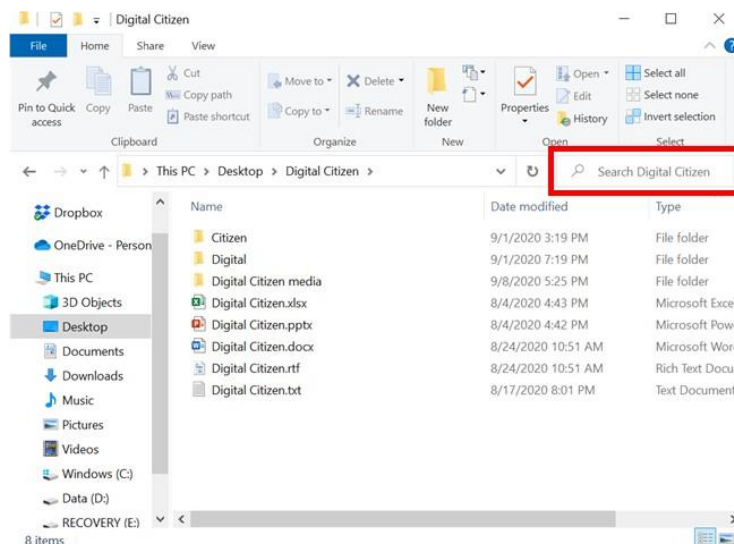
DAFTAR ISI

DAFTAR ISI	i
BAB I DESKRIPSI TUGAS	1
BAB II LANDASAN TEORI.....	5
BAB III ANALISIS PEMECAHAN MASALAH.....	9
BAB IV IMPLEMENTASI DAN PENGUJIAN.....	15
BAB V KESIMPULAN DAN SARAN.....	27
I. Kesimpulan.....	28
II. Saran.....	28
DAFTAR PUSTAKA.....	ii
LINK-LINK PENTING.....	ii

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 mencari 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.

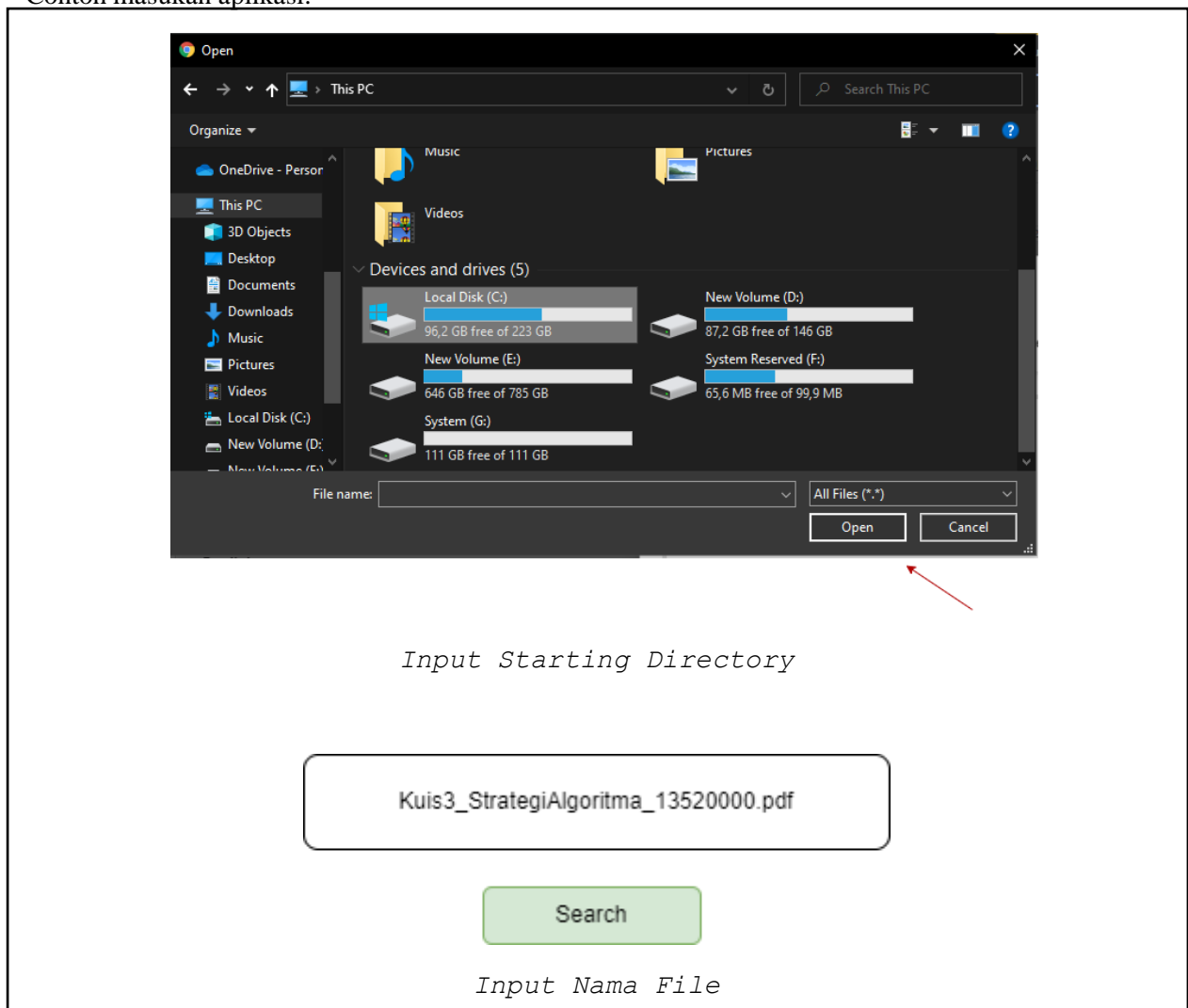
Dalam tugas besar ini, Anda akan diminta untuk membangun sebuah aplikasi GUI sederhana yang

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.

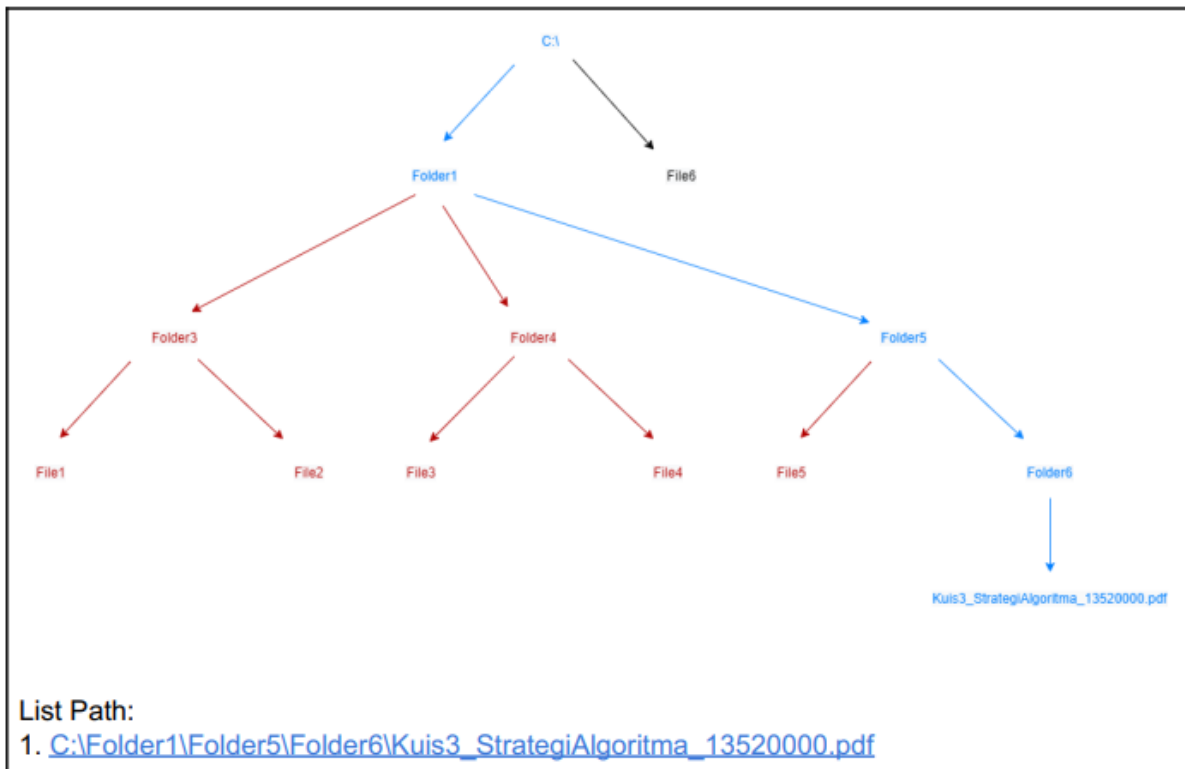
Contoh Input dan Output Program

Contoh masukan aplikasi:



Gambar 2. Contoh input program

Contoh output aplikasi:

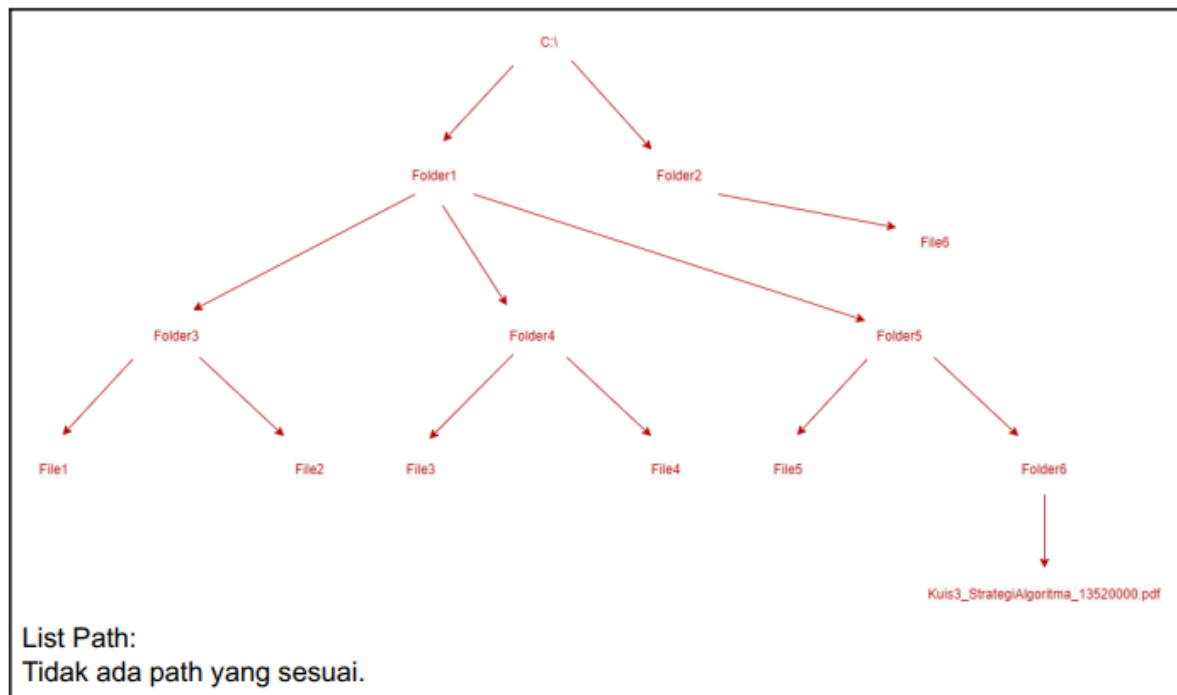


Gambar 3. Contoh output program

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

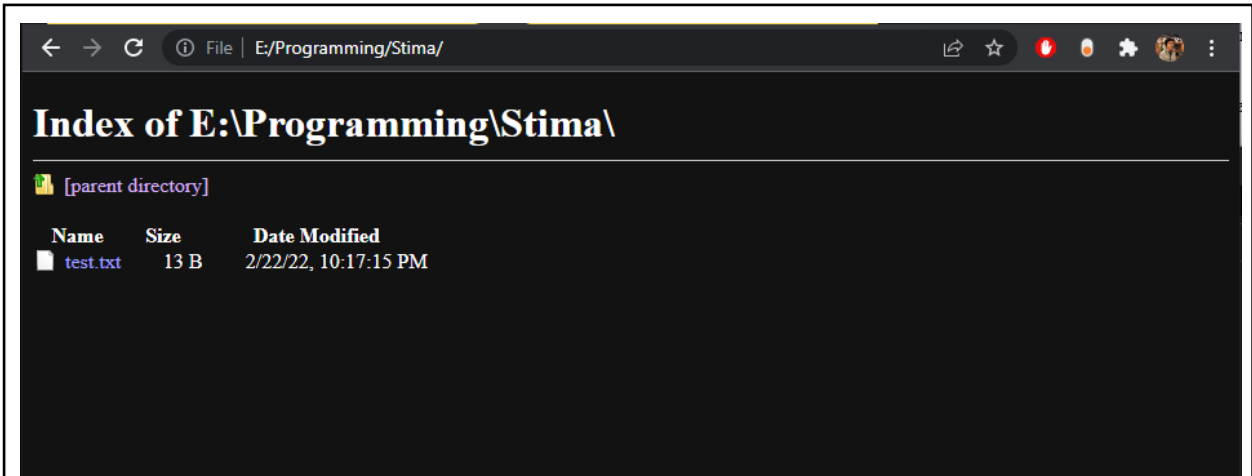
Jika file yang ingin dicari pengguna tidak ada pada direktori file, misalnya saat pengguna mencari Kuis3Probststat.pdf, maka path pencarian DFS adalah sebagai berikut:

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

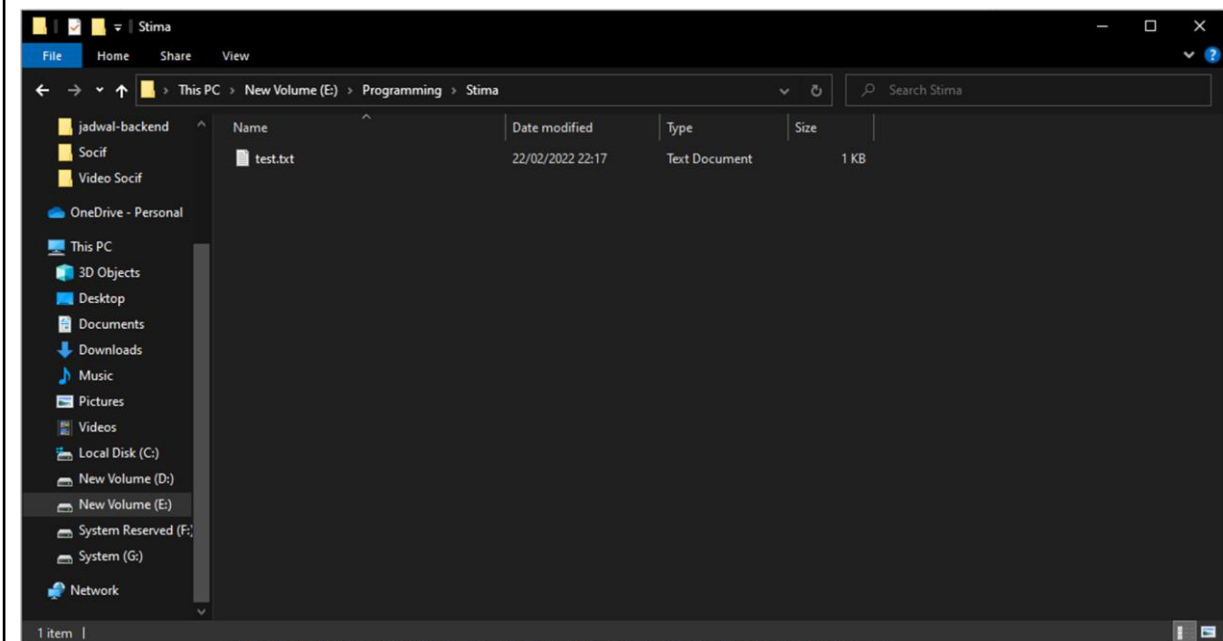
Pada gambar 4, semua simpul dan cabang berwarna merah yang menandakan seluruh direktori sudah selesai diperiksa semua namun tidak ada yang mengarah ke tempat file berada.

Contoh Hyperlink pada Path:

Gambar 4. Contoh ketika hyperlink di-klik



Contoh Hyperlink Dibuka Melalui Browser



Contoh Hyperlink Dibuka Melalui Browser

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

e.g. "word.pdf"

☐ Find all occurrence

Input Metode Pencarian

☐ BFS

☒ DFS

Search

Output

Input

Choose Starting Directory

Change Folder.. C:/

Input File Name

Kuis3_StrategiAlgoritma_13520000.pdf

☐ Find all occurrence

Input Metode Pencarian

☐ BFS

☒ DFS

Search

Output

```
graph TD; C1[C:\] --> Folder1[Folder1]; C1 --> File6[File6]; Folder1 --> Folder3[Folder3]; Folder1 --> Folder4[Folder4]; Folder1 --> Folder5[Folder5]; Folder3 --> File1[File1]; Folder3 --> File2[File2]; Folder4 --> File3[File3]; Folder4 --> File4[File4]; Folder5 --> File5[File5]; Folder5 --> Folder6[Folder6]; Folder6 --> Kuis3[Kuis3_StrategiAlgoritma_13520000.pdf];
```

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 design layout aplikasi dibebaskan dengan syarat mengandung seluruh input dan output yang terdapat pada spesifikasi.

Spesifikasi GUI:

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.

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

- 1) Buatlah program dalam bahasa **C#** untuk melakukan penelusuran *Folder Crawling* sehingga diperoleh hasil pencarian file yang diinginkan. Penelusuran harus memanfaatkan algoritma **BFS dan DFS**.
- 2) Awalnya program menerima sebuah input folder pada direktori yang ada dan nama file yang akan dicari oleh program.
- 3) Terdapat dua pilihan pencarian, yaitu:
 - a. Mencari 1 file saja
Program akan memberhentikan pencarian ketika sudah menemukan file yang memiliki nama sama persis dengan input nama file.
 - b. Mencari semua kemunculan file pada folder root
Program akan berhenti ketika sudah memeriksa semua file yang terdapat pada folder root dan program akan menampilkan daftar semua rute file yang memiliki nama sama persis dengan input nama file
- 4) Program kemudian dapat menampilkan **visualisasi pohon pencarian file** berdasarkan informasi direktori dari folder yang di-input. Pohon hasil pencarian file ini memiliki root adalah folder yang di-input dan setiap daunnya adalah file yang ada di folder root tersebut. Setiap folder/file direpresentasikan sebagai sebuah node atau simpul pada pohon. Cabang pada pohon menggambarkan folder/file yang terdapat di folder *parent*-nya.

Visualisasi pohon juga harus disertai dengan **keterangan** node yang sudah diperiksa, node yang sudah masuk antrian tapi belum diperiksa, dan node yang bagian dari rute hasil penemuan.

Proses visualisasi ini boleh memanfaatkan pustaka atau kaskas yang tersedia. Sebagai referensi, salah satu kaskas yang tersedia untuk melakukan visualisasi adalah **MSAGL** (<https://github.com/microsoft/automatic-graph-layout>) Berikut ini adalah panduan singkat terkait penggunaan MSAGL oleh tim asisten yang dapat diakses pada:

<https://docs.google.com/document/d/1XhFSpHU028Gaf7YxkmdbluLkOgVI3MY6gt1t-PL30LA/edit?usp=sharing>

- 5) Program juga dapat menyediakan *hyperlink* pada setiap hasil rute yang ditemukan. *Hyperlink* ini akan membuka folder parent dari file yang ditemukan. Folder hasil *hyperlink* dapat dibuka dengan *browser* atau *file explorer*.
- 6) Mahasiswa **tidak diperkenankan** untuk melihat atau menyalin library lain yang mungkin tersedia bebas terkait dengan pemanfaatan BFS dan DFS. Tapi untuk algoritma lainnya seperti *string matching* dan akses *directory*, diperbolehkan menggunakan library jika ada.

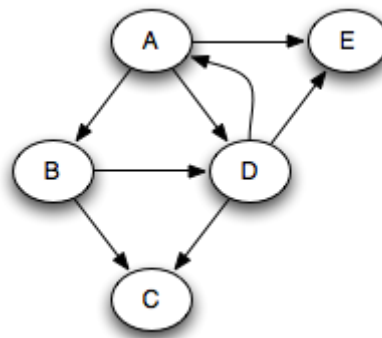
BAB II

LANDASAN TEORI

2.1 Dasar Teori Graf, BFS, dan DFS

2.1.1 Graf Traversal

Graf traversal merupakan proses mengunjungi setiap simpul yang terdapat pada Graf terhubung secara sistematis. Graf digambarkan sebagai representasi dari masalah yang hendak kita cari solusinya dengan mengunjungi simpul-simpul yang ada.



Gambar 2.1 Graf traversal

(sumber: <https://www.google.com/>)

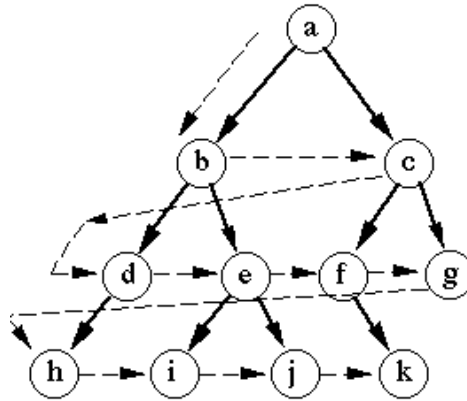
Umumnya, pencarian graph traversal dibagi menjadi dua, yaitu **Breadth First Search (BFS)** dan **Depth First Search (DFS)**. *Breadth First Search* seperti namanya, menjalankan pencarian melebar terlebih dahulu, sedangkan *Depth First Search* menjalankan pencarian mendalam. Ada banyak kegunaan dari graph traversal ini yaitu:

- *Citation map*
- *Web spider*
- Pencarian *file* pada *folder directory*
- Dsb.

2.1.2 BFS (Breadth-First Search)

BFS adalah algoritma pencarian pada graf dimana pencarian dilakukan dengan mengunjungi simpul tetangga terlebih dahulu sebelum simpul anaknya. Hal ini menyebabkan BFS terjadi secara menyamping baru kebawah. Secara umum, dalam pengimplementasiannya, algoritma BFS tidak perlu menggunakan kode yang rekursif, walaupun tidak menutup kemungkinan menggunakan algoritma rekursif. Pencarian BFS ini memanfaatkan struktur data *Queue* karena pengimplementasiannya menggunakan prinsip FIFO (*First In First Out*). Apabila graf berbentuk pohon berakar, maka semua simpul pada level d harus dikunjungi sebelum mengunjungi semua simpul pada level $d+1$. Kompleksitas

waktu dari BFS adalah $O(V + E)$ apabila digunakan list ketetanggaan, dan $O(V^2)$ apabila digunakan matriks ketetanggaan, dengan V sebagai simpul dan E sebagai sisi.

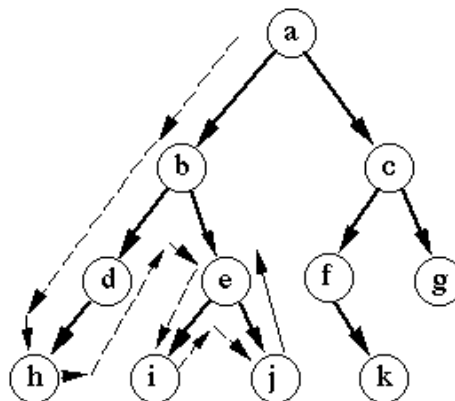


Gambar 2.2 Pencarian secara BFS

(sumber: <https://www.google.com/>)

2.1.3 DFS (Depth-First Search)

DFS merupakan algoritma pencarian pada graf dimana pencarian dilakukan dengan mengunjungi semua simpul anak hingga level terdalamnya terlebih dahulu baru mengunjungi simpul disampingnya. Dalam algoritma DFS ini, umumnya kita memanfaatkan metode rekursi dimana terus mengiterasi anak dari root sampai ke node yang tidak memiliki anak lagi (daun). Pencarian BFS ini juga memanfaatkan struktur data Stack karena pengimplementasiannya menggunakan prinsip LIFO (*Last In First Out*). Kompleksitas waktu dari DFS adalah $O(V + E)$ apabila digunakan list ketetanggaan, dan $O(V^2)$ apabila digunakan matriks ketetanggaan, dengan V sebagai simpul dan E sebagai sisi.



Gambar 2.3 Pencarian secara DFS

(sumber: <https://www.google.com/>)

2.2 C# Desktop application development

C# Desktop application development merupakan sebuah program yang dapat kita manfaatkan untuk berbagai macam kebutuhan untuk pengembangan perangkat lunak. Framework yang dapat digunakan untuk C# Desktop Application Development adalah Visual Studio, ASP.NET Web Matrix, SharpDevelop, dan lain sebagainya. Aplikasi ini dapat kita gunakan tanpa menggunakan akses internet. Visual Studio dapat dijalankan di sistem operasi Windows dan sistem operasi MacOS. Dalam tugas besar kali ini, salah satu framework yang digunakan merupakan .Net. Framework ini sangat memudahkan programmer karena bentuk visual dari aplikasi langsung ditampilkan sehingga memudahkan dalam proses penyusunan perangkat lunak.

BAB III

ANALISIS PEMECAHAN MASALAH

3.1 Langkah-Langkah Pemecahan masalah

Dalam pemecahan permasalahan tugas besar kali ini, kelompok kami mengawali pengerjaan dengan melakukan analisis permasalahan untuk memudahkan pengerjaan. Pada tugas besar kali ini, dibutuhkan suatu model dari fitur pencarian *file* pada suatu *starting directory* dengan menggunakan teknik *folder crawling*. Analisis terhadap permasalahan tersebut menghasilkan beberapa bagian yang dapat dijabarkan sebagai berikut:

1. Membuat GUI dalam bentuk Windows Forms *app* yang dapat menerima *input* berupa *path* dari *starting directory*, nama *filename* yang dicari, *speed interval* yang menentukan kecepatan pembuatan graf, pilihan jumlah kemunculan, dan pilihan metode pencarian antara BFS atau DFS. Pilihan metode pencarian akan menentukan jenis metode yang dilakukan untuk mengiterasi *folder*, sedangkan *starting directory*, *filename*, dan pilihan jumlah kemunculan akan diteruskan sebagai argumen pada fungsi pencarian (BFS/DFS).
2. Membuat fungsi pencarian yang menerapkan algoritma BFS serta fungsi pencarian yang menerapkan algoritma DFS. Kedua fungsi tersebut akan digunakan untuk menelusuri folder pada *directory*. Kami memutuskan untuk menggunakan struktur data *stack* untuk algoritma DFS dan *queue* untuk algoritma BFS untuk mempermudah dalam pencarian dan visualisasi yang akan dilakukan oleh aplikasi ini.
3. Membuat fungsi yang menangani visualisasi dari graf. *Graph visualizer* akan menerima hasil dari algoritma BFS/DFS dan memvisualisasikan hasil dari pencarian *folder* dalam bentuk pohon. Pembuatan graf dilakukan secara progresif dengan kecepatan yang ditentukan oleh pengguna. Dalam implementasinya, kami memanfaatkan MSAGL sebagai *tool* untuk menentukan *layout* dan hasil visualisasi dari graf. Visualisasi dari graf akan ditampilkan dalam GUI dengan cara dimasukkan ke dalam suatu *panel* yang berada di dalam *forms*.

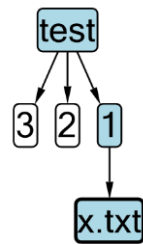
3.2 Proses *mapping* Persoalan Elemen BFS dan DFS

Program akan menerima inputan user berupa folder root tempat pencarian dimulai. Dari root folder ini akan dihasilkan graf yang menggambarkan proses penelusuran yang terjadi. Seperti yang kita ketahui, elemen penyusun graf merupakan simpul dan sisi. Dalam hal ini, simpul menggambarkan folder dan file yang kita telusuri dan sisi menggambarkan hierarki dari root folder dengan file ataupun folder yang berada di root folder. Setelah *mapping* permasalahan dengan graf, penelusuran akan dilakukan sesuai dengan algoritma pilihan dari user yaitu BFS atau DFS.

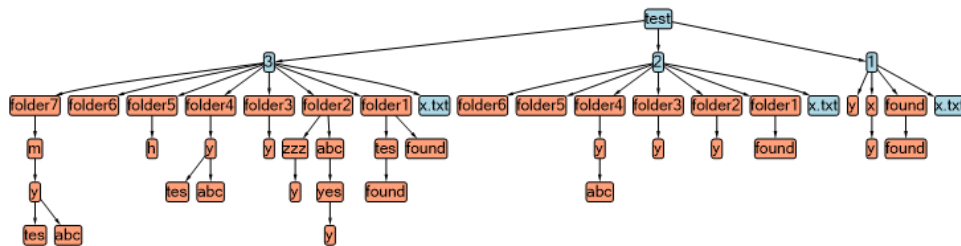
Hasil yang dikeluarkan dapat dipilih dengan pengeluaran semua hasil atau pengeluaran hanya hasil pertama. Apabila mode semua hasil, maka akan mengiterasi seluruh node yang ada sampai daun. Jika hanya satu hasil, maka jika ditemukan akan dilakukan pemberhentian pencarian. Hasil akan ditampilkan berupa warna biru sebagai path hasil dari pencarian, warna merah sebagai path yang telah ditelusuri, dan warna putih untuk path yang belum ditelusuri.

- Kasus 1: DFS, pencarian tidak ditemukan

- Kasus 5: BFS, pencarian ditemukan, one file



- Kasus 6: BFS, pencarian ditemukan, all occurrence



BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Program

Searching.cs

```
Function DFS(start:string, fileToSearch:string, checkAllOccur:bool) -> list of string
    found <- false
    stack <- empty vector of string
    output <- empty vector of string

    // Push main dir as the first stack element
    stack.Push(start)

    // loop until stack is empty
    while (stack.Count > 0) do
        // Initialize starting directory
        startDir <- stack.Pop()
        output.Add(startDir)

        // Find file
        files <- list of files in startDir;
        for each string file in files
            if (file contains fileToSearch and fileToSearch == filename)
                output.Add(file)
                found <- true
                output.Add("?") // separator antarpah

        // File found and x all occur
        if (found and not(checkAllOccur))
            // Add unvisited stack to the output list
            count <- stack.Count
            i <- 0
            repeat
                output.Add(stack.Pop())
                i <- i + 1
            until i < count
            break

        // Push all directories inside starting directory
        otherDir <- list of directories inside startDir
        for each string dir in otherDir
            stack.Push(dir)

    -> output.ToArray()

Function BFS(start:string, fileToSearch:string, checkAllOccur:bool) -> list of string
    found <- false
    queue <- empty vector of string
    output <- empty vector of string
```

```
queue.Enqueue(start)

// loop until queue is empty
while (queue.Count > 0) do
    // Initialize starting directory
    startDir <- queue.Dequeue()
    output.Add(startDir)

    // find file
    files = list of files in startDir
    for each string file in files
        if (file contains fileToSearch and fileToSearch == filename)
            output.Add(file)
            found <- true
            output.Add("?")

    // File found and x all occur
    if (found and not(checkAllOccur))
        // Add unvisited queue to the output list
        count <- queue.Count()
        i <- 0
        repeat
            output.Add(queue.Dequeue())
            i <- i + 1
        until i < count
        break

    // Enqueue all directories inside starting directory
    otherDir <- list of directories in startDir
    for each string dir in otherDir
        queue.Enqueue(dir)

-> output.ToArray()
```

Form1.cs

Procedure handleGraph(input/output Result : string[], input checkAllOccur : bool, input BFS : bool)

```
//clear form's listbox and hyperlinks
this.h11.Clear()
this.listBox1.Items.Clear()

// disable related buttons, trackbar, & hyperlink
this.searchBFS.Enabled <- false
this.searchDFS.Enabled <- false
this.trackBar1.Enabled <- false
this.listBox1.Visible <- true
this.listBox1.Enabled <- true
this.algorithmTime.Text <- "0"

// initialize graphviewer & panel display
this.graphPanel.Controls.Clear()
```

```
this.graphPanel.BackColor <- Color.White

viewer <- new microsoft MSAGL graph viewer
viewer.OutsideAreaBrush <- System.Drawing.Brushes.White
viewer.Size <- this.graphPanel.Size

// create a graph object
graph <- new microsoft MSAGL drawing graph

// startpath = main directory
startPath <- Result[0];

// loop
i <- 1
repeat
  // if result[i] is ?
  if (Result[i] == "?")
    // path is before ?
    foundfile <- Result[i - 1]

    // add hyperlink inside listbox
    hyperlinks link = hyperlinks
    link.hyperlink_name <- foundfile
    link.hyperlink_link <- foundfile
    listBox1.Items.Add(foundfile)
    hll.Add(link)

    // change path color
    graph.FindNode(startPath).FillColor <- LightBlue
    graph.FindNode(foundfile).FillColor <- LightBlue
    graph.FindNode(foundfile).LineWidth <- 2
    parent <- directory name of foundfile
    // loop to the main directory
    while (parent != startPath) do
      graph.FindNode(parent).FillColor <- LightBlue
      parent <- directory name of parent

// not all occur and file has been found
if (not(checkAllOccur) and Result[i] == "?")
{
  // print unvisited directory
  j = i + 1
  repeat
    string folderuv = Result[j];
    string rootuv = Path.GetDirectoryName(folderuv);
    graph.AddEdge(rootuv, folderuv); // Create node
    if (graph.FindNode(folderuv).FillColor != LightBlue and
graph.FindNode(folderuv).FillColor != LightSalmon)
      else
        graph.FindNode(folderuv).Attr.FillColor <- White
        graph.FindNode(folderuv).Label.Text = directory of folderuv
graph.FindNode(rootuv).Label.Text = directory of rootuv
        if (graph.FindNode(rootuv).FillColor != LightBlue &&
graph.FindNode(rootuv).FillColor != lightSalmon)
```

```
graph.FindNode(rootuv).FillColor = White
    j <- j + 1
    until j < Result.length

    // show
    viewer.Graph = graph
    this.graphPanel.SuspendLayout()
    viewer.Dock = DockStyle.Fill
    this.graphPanel.Controls.Add(viewer)
    // quit
    break
}

// find all occur and result[i] is ?, lanjut
if (checkAllOccur && Result[i] == "?")
    continue

// create new edge
if (i != 0) await Task.Delay(this.trackBar1.Value)
folder <- Result[i]
root <- directory name of folder
graph.AddEdge(root, folder) // Create node
if (graph.FindNode(folder).FillColor != LightBlue)
    graph.FindNode(folder).FillColor <- LightSalmon;
graph.FindNode(folder).Label.Text <- directory name of folder
graph.FindNode(root).Label.Text <- directory name of root
if (graph.FindNode(root).FillColor != LightBlue)
    graph.FindNode(root).FillColor <- LightSalmon

// show
viewer.Graph <- graph
this.graphPanel.SuspendLayout()
viewer.Dock <- DockStyle.Fill
this.graphPanel.Controls.Add(viewer)
i <- i + 1
until i < Result.length
this.graphPanel.ResumeLayout()

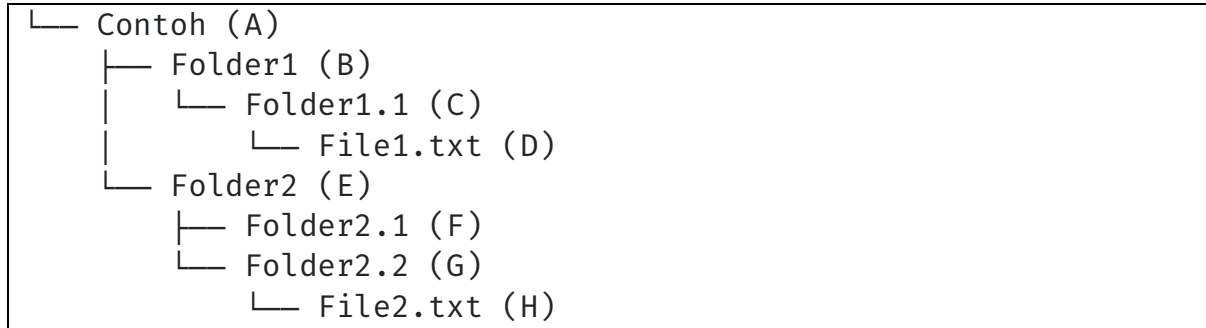
// reset view
this.trackBar1.Enabled <- true
this.searchBFS.Enabled <- true
this.searchDFS.Enabled <- true
this.searchDFS.BackColor <- Transparent
this.searchBFS.BackColor <- Transparent
}
```

4.2 Struktur Data

Terdapat beberapa struktur data yang digunakan pada program, yaitu :

1. Stack

Stack merupakan struktur data yang menggunakan prinsip LIFO (*Last In First Out*) dimana penggunaannya menggunakan fungsi push (untuk memasukkan elemen) dan pop (untuk mengeluarkan elemen). Struktur data ini diimplementasikan pada algoritma DFS.



Gambar 4.2 Contoh Hierarki Folder

Apabila mengacu pada gambar 4.2, maka proses iterasi untuk mencari File2 jika memanfaatkan *Stack* adalah sebagai berikut :

Iterasi 1	Simpul Ekspan : A Simpul Hidup : B _A , E _A
Iterasi 2	Simpul Ekspan : B _A Simpul Hidup : C _{BA} , E _A
Iterasi 3	Simpul Ekspan : C _{BA} Simpul Hidup : E _A
Iterasi 4	Simpul Ekspan : E _A Simpul Hidup : F _{EA} , G _{EA}
Iterasi 5	Simpul Ekspan : F _{EA} Simpul Hidup : G _{EA}
Iterasi 6	Simpul Ekspan : G _{EA} → file ditemukan Simpul Hidup : -

2. Queue

Queue merupakan struktur data yang menggunakan prinsip FIFO (*First In First Out*) dimana penggunaannya menggunakan fungsi enqueue (untuk memasukkan elemen) dan dequeue (untuk mengeluarkan elemen). Struktur data ini diimplementasikan pada algoritma BFS. Apabila mengacu pada gambar 4.2, maka proses iterasi untuk mencari File2 jika memanfaatkan *Queue* adalah sebagai berikut :

Iterasi 1	Simpul Ekspan : A Simpul Hidup : B _A , E _A
Iterasi 2	Simpul Ekspan : B _A Simpul Hidup : E _A , C _{BA}
Iterasi 3	Simpul Ekspan : E _A Simpul Hidup : C _{BA} , F _{EA} , G _{EA}
Iterasi 4	Simpul Ekspan : C _{BA} Simpul Hidup : F _{EA} , G _{EA}

Iterasi 5	Simpul Ekspan : F_{EA} Simpul Hidup : G_{EA}
Iterasi 6	Simpul Ekspan : $G_{EA} \rightarrow$ file ditemukan Simpul Hidup : -

3. List

List merupakan struktur data yang menyimpan elemen atau objek dalam bentuk list. Terdapat 2 cara untuk menambahkan elemen ke dalam list, yaitu Add dan AddRange, dimana Add digunakan untuk menambahkan satu elemen dan AddRange digunakan untuk menambahkan beberapa elemen secara bersamaan ke dalam List. Struktur data ini digunakan untuk menyimpan kumpulan *path* yang diselidiki dan

4. Array

Array adalah struktur data yang menyimpan beberapa elemen dengan tipe yang sama dalam suatu lokasi memori berdekatan. Penggunaan array ini dilakukan untuk menyimpan hasil pencarian BFS dan juga DFS.

4.3 Tata Cara Penggunaan Program

Bentuk program ketika baru dibuka dan fitur yang tersedia adalah sebagai berikut :

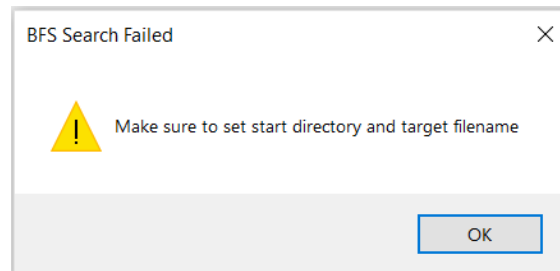


1. *Button* yang mengarahkan pengguna ke *file explorer* untuk memilih direktori folder yang diinginkan
2. *Textbox* otomatis (tidak menerima masukan) yang akan menampilkan path dari direktori folder yang telah dipilih pada no 1
3. *Textbox* yang menerima nama file yang ingin dicari

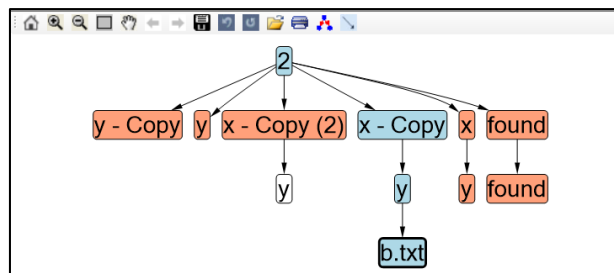
4. *Checkbox* yang di-*tick* apabila ingin mengiterasi seluruh folder untuk mencari semua file yang sesuai. Apabila tidak di-*tick*, pencarian berhenti pada penemuan file pertama yang sesuai.

5. *trackBar* yang digunakan untuk menerima input berupa nilai interval pembentukan graf yang diinginkan

6. *Button* untuk memilih jenis pencarian (BFS/DFS). Pencarian hanya akan dilakukan apabila direktori folder dan nama file yang diinginkan sudah terisi. Apabila belum, maka akan muncul message box seperti berikut :



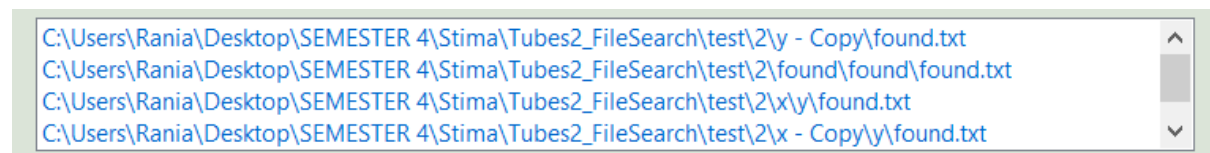
7. Panel untuk menampilkan hasil pembuatan graf, contoh hasilnya adalah sebagai berikut :



- Warna orange : Direktori yang sudah dikunjungi
- Warna biru : Direktori yang memiliki file yang sesuai
- Warna putih : Direktori yang ada di antrian tapi belum dikunjungi

8. Teks yang menampilkan lama program berjalan untuk mengiterasi sebuah direktori dan mencari file yang sesuai (tidak termasuk waktu pembuatan graf).

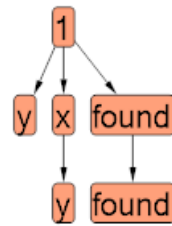
9. *ListBox* yang menampilkan *hyperlink* dari file yang sesuai, Ketika salah satu link yang terdapat dalam *ListBox* ditekan, maka file akan terbuka. *ListBox* ini baru muncul ketika sudah dilakukan pencarian. Contoh hasilnya adalah sebagai berikut:



4.4 Pengujian

Pengujian 1 (Folder test/1)

Hierarki Folder



Nama File found.txt

Hasil Uji

BFS allOccur

RESULT

Algorithm Time 1 ms

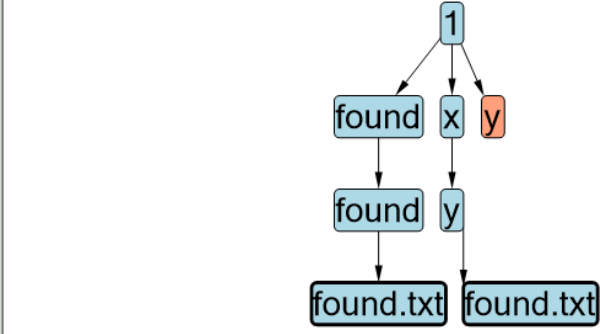
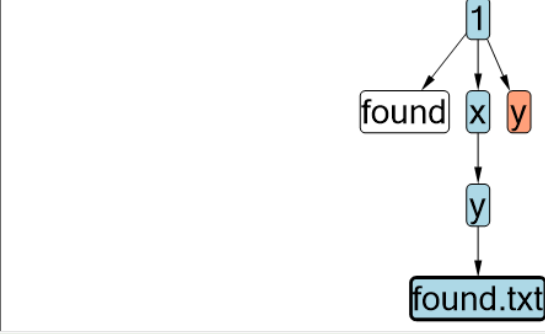
C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\1\found\found\found.txt
C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\1\x\y\found.txt

BFS x allOccur

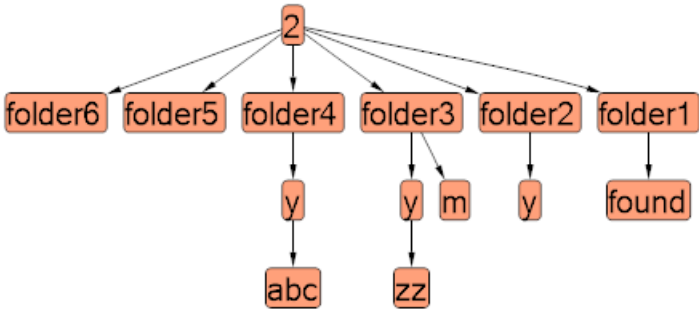
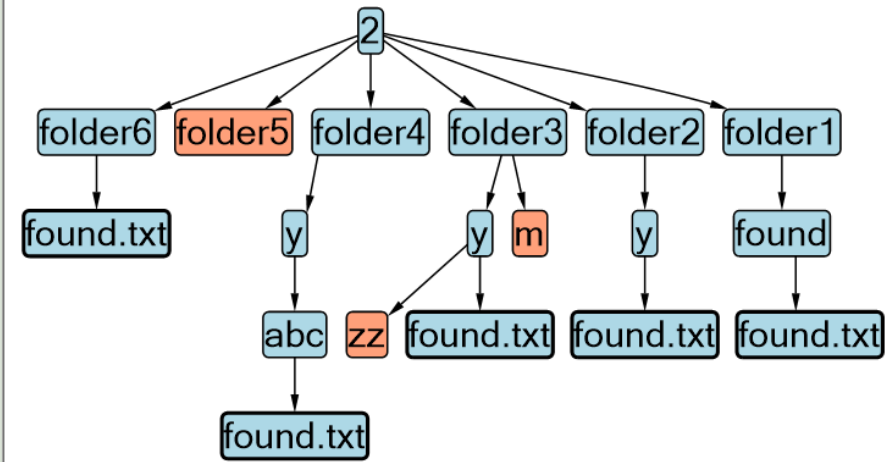
RESULT

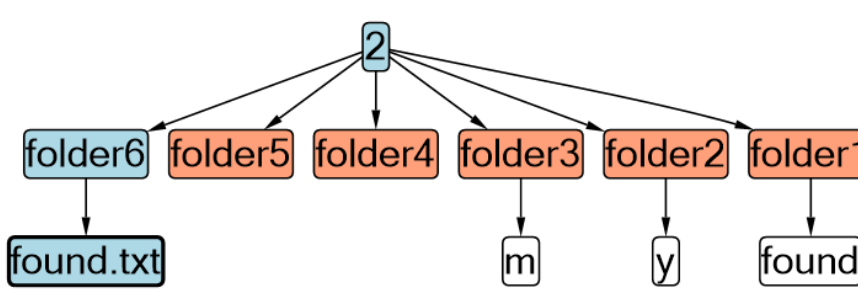
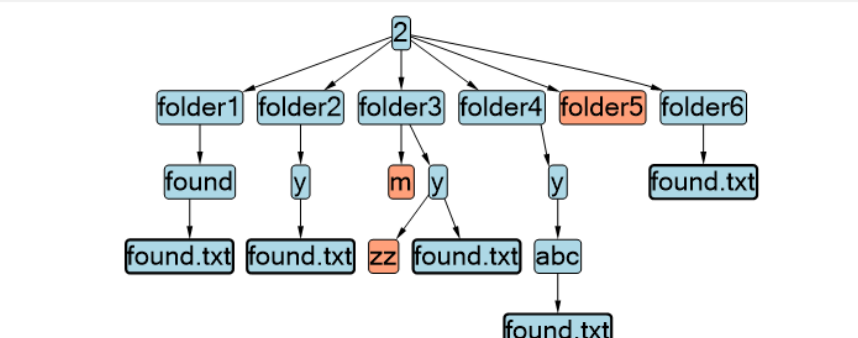
Algorithm Time 0 ms

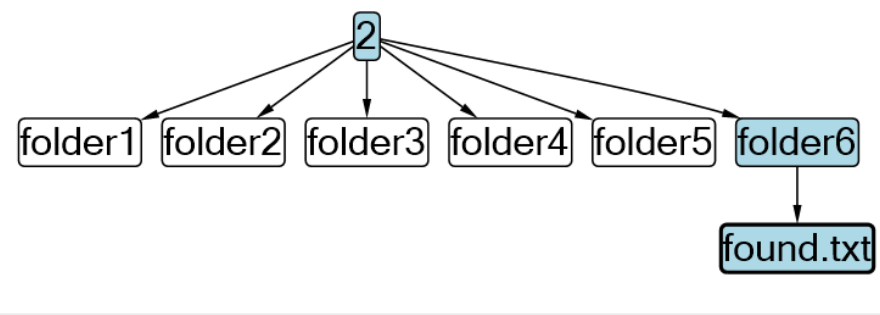
C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\1\found\found\found.txt

<p>DFS <i>allOccur</i></p>	<div> <p>RESULT</p>  <p>Algorithm Time 2 ms</p> <p>C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\1\x\y\found.txt C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\1\found\found\found.txt</p> </div>
<p>DFS <i>x allOccur</i></p>	<div> <p>RESULT</p>  <p>Algorithm Time 1 ms</p> <p>C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\1\x\y\found.txt</p> </div>
<p>Analisis</p> <p>Pada kasus tidak <i>allOccur</i> (tidak mengiterasi seluruh subdirektori) dan <i>allOccur</i>, waktu algoritma yang dihasilkan BFS lebih cepat 1 ms dibanding DFS. Oleh sebab itu, dapat disimpulkan bahwa pada kasus ini, BFS lebih efisien dibandingkan DFS.</p>	

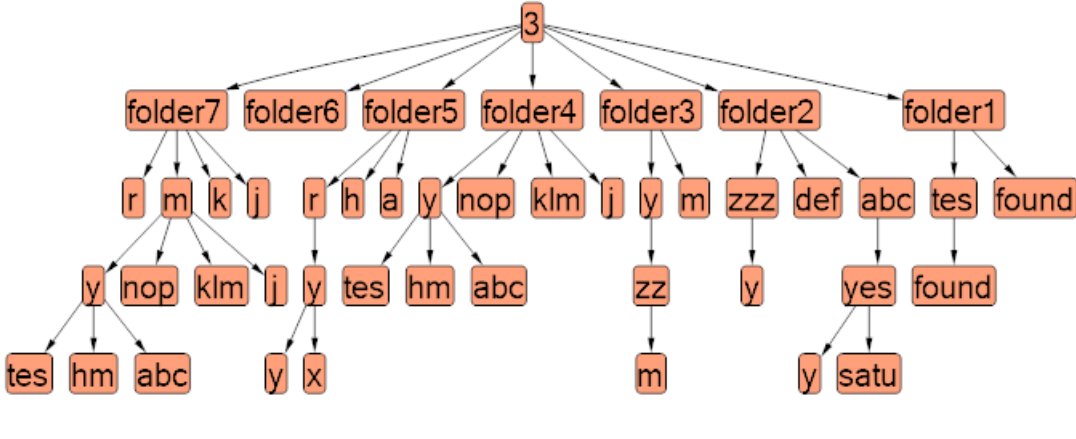
Pengujian 2 (Folder test/2)

Hierarki Folder	
	
Nama File	found.txt
Hasil Uji	
BFS allOccur	<div> <p>RESULT</p>  <p>Algorithm Time 1 ms</p> <p>C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\2\folder6\found.txt C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\2\folder1\found\found.txt C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\2\folder2\y\found.txt C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\2\folder3\y\found.txt</p> </div>

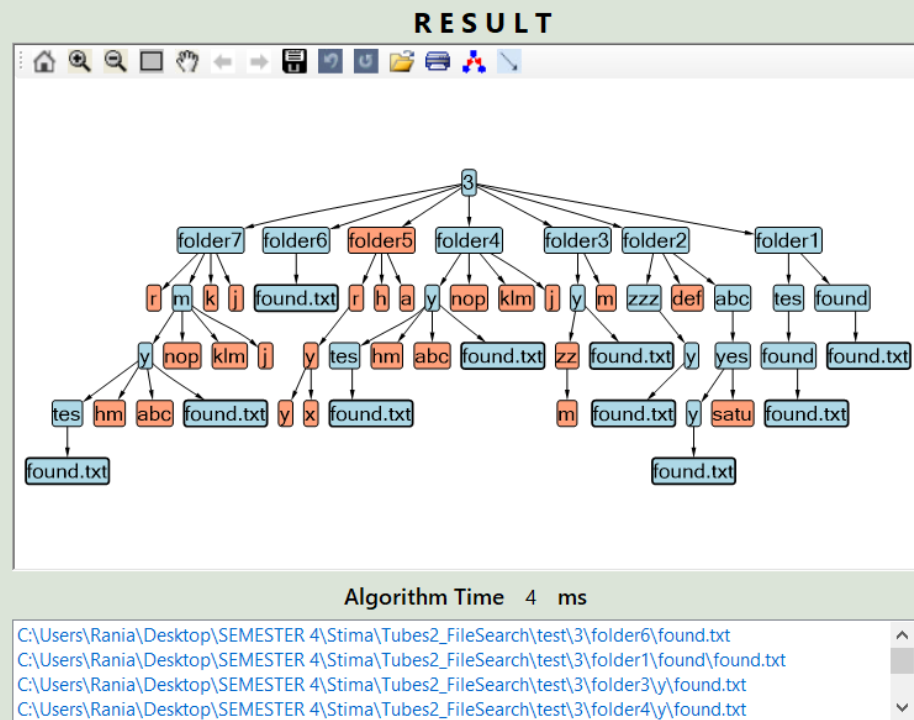
<p>BFS <i>x allOccur</i></p>	<div data-bbox="459 286 1375 1003"> <h3>RESULT</h3>  <p>Algorithm Time 1 ms</p> <p>C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\2\folder6\found.txt</p> </div>
<p>DFS <i>allOccur</i></p>	<div data-bbox="459 1034 1375 1612"> <h3>RESULT</h3>  <p>Algorithm Time 3 ms</p> <p>C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\2\folder6\found.txt C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\2\folder4\y\abc\found.txt C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\2\folder3\y\found.txt C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\2\folder2\y\found.txt</p> </div>

DFS x <i>allOccur</i>	<div data-bbox="459 277 1364 857"> <p style="text-align: center;">RESULT</p>  <p style="text-align: center;">Algorithm Time 0 ms</p> <p style="text-align: center;">C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\2\folder6\found.txt</p> </div>
Analisis	
<p>Pada kasus tidak <i>allOccur</i> (tidak mengiterasi seluruh subdirektori), waktu algoritma yang dihasilkan DFS lebih cepat 1 ms dibanding BFS. Namun, ketika mengiterasi seluruh subdirektori, waktu algoritma yang dihasilkan BFS lebih cepat 2 ms dibanding DFS. Oleh sebab itu, dapat disimpulkan bahwa pada kasus ini, BFS dan DFS memiliki efisiensi yang hampir sama.</p>	

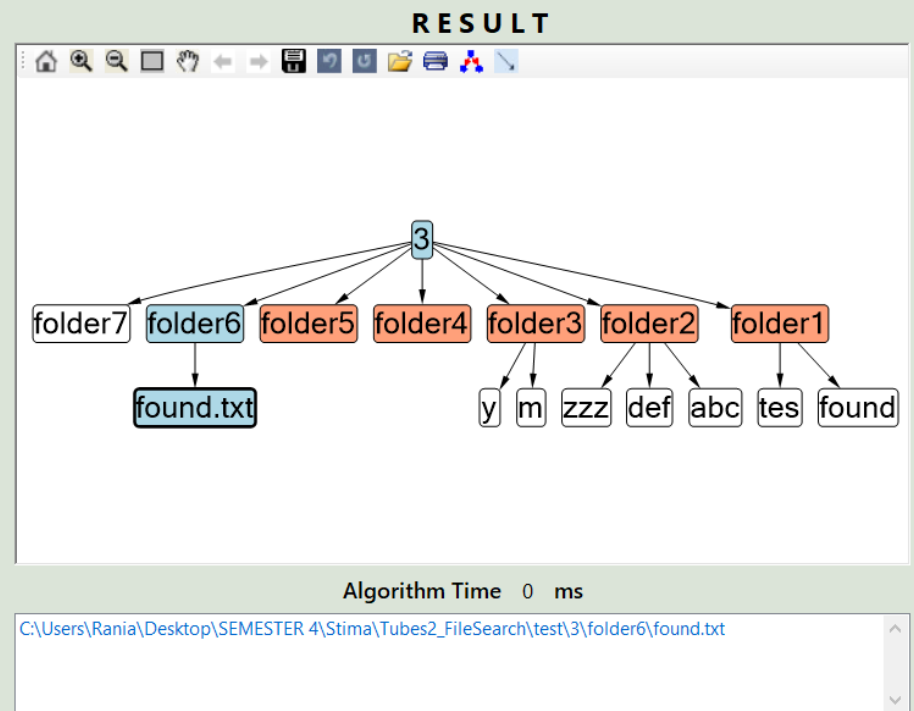
Pengujian 3 (Folder test/3)

Hierarki Folder	
	
Nama File	found.txt
Hasil Uji	

BFS *allOccur*

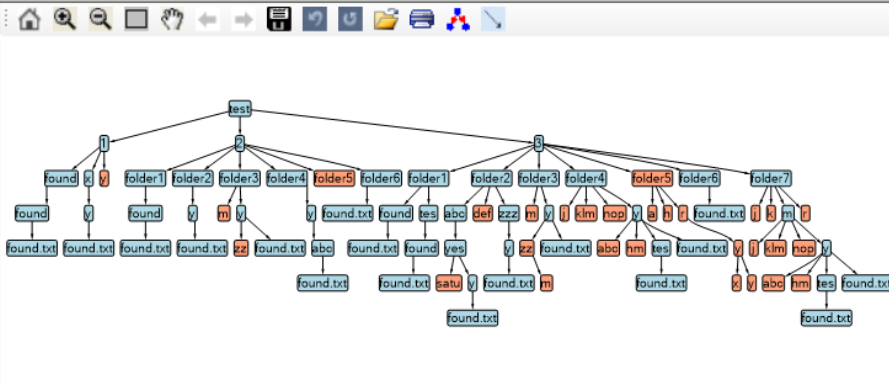
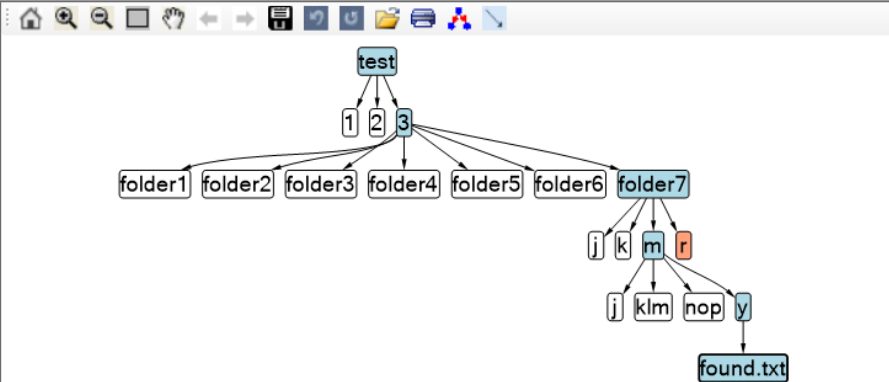


BFS *x allOccur*



Pengujian 4 (Folder test)

Hierarki Folder	
Nama File	found.txt
Hasil Uji	
BFS allOccur	<p>RESULT</p> <p>Algorithm Time 7 ms</p> <p>C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\2\folder6\found.txt C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\3\folder6\found.txt C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\1\found\found\found.txt C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\1\x\y\found.txt</p>
BFS x allOccur	<p>RESULT</p> <p>Algorithm Time 1 ms</p> <p>C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\2\folder6\found.txt</p>

<p>DFS <i>allOccur</i></p>	<div data-bbox="459 277 1374 864"> <p style="text-align: center;">RESULT</p>  <p style="text-align: center;">Algorithm Time 6 ms</p> <div style="border: 1px solid black; padding: 5px;"> <p>C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\3\folder7\m\y\found.txt</p> <p>C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\3\folder7\m\y\tes\found.txt</p> <p>C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\3\folder6\found.txt</p> <p>C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\3\folder4\y\found.txt</p> </div> </div>
<p>DFS <i>x allOccur</i></p>	<div data-bbox="459 887 1374 1473"> <p style="text-align: center;">RESULT</p>  <p style="text-align: center;">Algorithm Time 0 ms</p> <div style="border: 1px solid black; padding: 5px;"> <p>C:\Users\Rania\Desktop\SEMESTER 4\Stima\Tubes2_FileSearch\test\3\folder7\m\y\found.txt</p> </div> </div>
<p style="text-align: center;">Analisis</p> <p>Pada kasus tidak allOccur (tidak mengiterasi seluruh subdirektori) maupun allOccur, waktu algoritma yang dihasilkan DFS lebih cepat 1 ms dibanding BFS. Oleh sebab itu, dapat disimpulkan bahwa pada kasus ini, DFS lebih efisien dibandingkan BFS.</p>	

BAB V

KESIMPULAN DAN SARAN

I. Kesimpulan

Algoritma BFS dan DFS merupakan algoritma yang sering dipakai di kehidupan kita sehari-hari. Pencarian secara BFS dilakukan menyamping, sedangkan DFS secara mendalam. Contoh penggunaannya adalah dengan mengimplementasikannya pada proses penemuan suatu file yang terdapat dalam folder. Dalam tugas besar kali ini, kelompok kami telah berhasil mengimplementasikan algoritma BFS dan DFS dalam proses *file searching*. Pemilihan metode yang digunakan dapat sesuai dengan pengguna dengan pemilihan algoritma pencarian BFS jika ingin mendapatkan jalur paling singkat, sedangkan algoritma pencarian DFS untuk pemanfaatan memori yang efisien.

II. Saran

Saran untuk pengerjaan tugas besar kali ini diantaranya:

- Pembelajaran lebih lanjut mengenai C# desktop development dan library MSAGL
- Memperhatikan spesifikasi laptop yang sesuai untuk mengunduh visual studio
- Meningkatkan kualitas UI menjadi lebih modern

REFERENSI

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

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf> (diakses pada 18 Maret 2022)

<https://www.geeksforgeeks.org/c-sharp-stack-class/> (diakses pada 18 Maret 2022)

<https://www.geeksforgeeks.org/c-sharp-queue-class/> (diakses pada 18 Maret 2022)

<https://docs.microsoft.com/en-us/dotnet/csharp/> (diakses pada 18 Maret 2022)

LINK-LINK PENTING

Link repository github: https://github.com/graceclaudia19/Tubes2_FileSearch

Link video: (terdapat juga di readme): <http://bit.ly/DagorlzTheExplorer>