Laporan Tugas Besar II
IF2211 Strategi Algoritma
Pengaplikasian Algoritma BFS dan DFS dalam Implementasi *Folder Crawling*

Oleh:

Ghebyon Tohada Nainggolan	13520079
Adzka Ahmadetya Zaidan	13520127
Thirafi Najwan Kurniatama	13520157

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

SEMESTER II 2021/2022

Daftar Isi

BAB I
DESKRIPSI TUGAS

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

Selain pohon, Anda diminta juga menampilkan list path dari daun-daun yang bersesuaian dengan hasil pencarian. Path tersebut diharuskan memiliki hyperlink menuju folder parent dari file yang dicari, agar file langsung dapat diakses melalui browser atau file explorer.

1.1 Spesifikasi GUI

Aplikasi yang akan dibangun dibuat berbasis GUI dengan spesifikasi GUI sebagai berikut:

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.

1.2 Spesifikasi Wajib

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 kakas yang tersedia. Sebagai referensi, salah satu kakas 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/1XhFSpHU028Gaf7YxkmdbluLkQgVl3MY6gt1tPL30LA/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 *Graph* Traversal

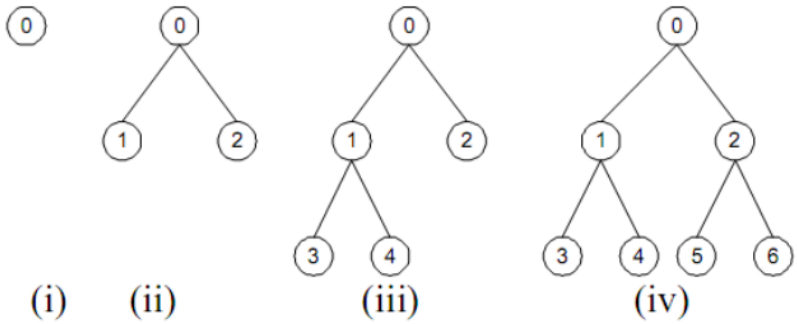
Graph Traversal merupakan algoritma mengunjungi simpul secara sistematik. Dengan asumsi, graf yang dilakukan algoritma ini merupakan graf terhubung. Algoritma pencarian solusi dapat berdasarkan informasi yang diberikan: *uninformed/blind search*–tidak ada informasi tambahan–atau *informed search*–pencarian berbasis heuristik, yaitu mengetahui non-goal state yang lebih menguntungkan (menjanjikan) dibanding yang lain. Pengunjungan sistematik yang dilakukan terhadap graf dapat berupa pencarian secara melebar–*Breadth First Search* (BFS)–atau pencarian secara mendalam–*Depth First Search (DFS)*–untuk pencarian solusi tanpa informasi tambahan.

2.2 Graf Statis dan Graf Dinamis

Graf statis merupakan graf yang sudah terbentuk sebelum proses pencarian dilakukan. Graf ini direpresentasikan sebagai struktur data. Sedangkan graf dinamis merupakan graf yang baru mulai dibentuk seiring proses pencarian dijalankan. Algoritma DFS dan BFS merupakan algoritma pencarian yang dapat digunakan pada graf dinamis untuk mencari suatu solusi.

2.3 *Breadth First Search* (BFS)

Algoritma *Breadth First Search* merupakan algoritma untuk menyelesaikan persoalan dengan melakukan pencarian solusi. Pencarian solusi dilakukan dengan pembentukan graf dinamis secara melebar. Kemudian, dilakukan pengecekkan untuk setiap simpul apakah simpul tersebut merupakan solusinya atau bukan. Setelah itu dilanjutkan dengan pembuatan simpul-simpul pada kedalaman selanjutnya.
Contoh BFS untuk pembentukan ruang status:



Gambar 2.3.1 Pohon Pembentukan Ruang Satus dengan BFS

Breadth First Search memiliki aspek pencarian sebagai berikut:

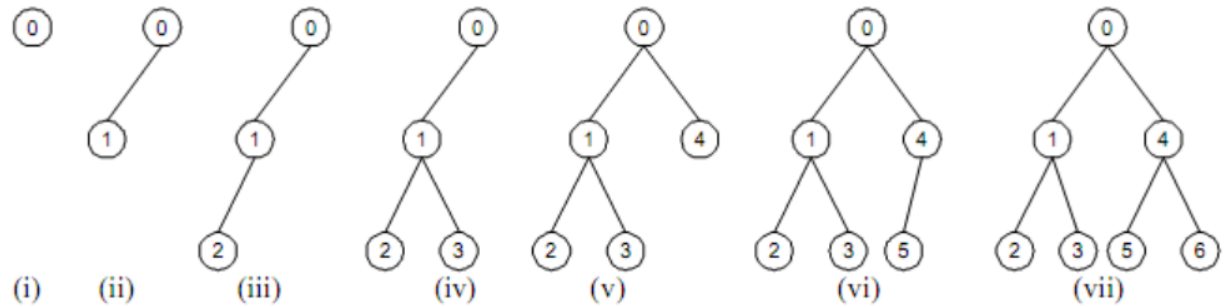
1. *Completeness,* menjamin ditemukannya solusi jika memang ada, selama memiliki suatu maksimum pencabangan yang mungkin terjadi pada suatu simpul

2. *Optimality,* menjamin solusi paling optimal jika jumlah langkah paling sedikit merupakan solusi optimalnya.
3. *Time Complexity,* dapat memakan waktu yang relatif lama jika solusi ada di kedalaman yang dalam.
4. *Space Complexity,* kurang baik dalam kompleksitas ruang karena dapat memakan memori dan ruang yang cukup banyak.

2.4 *Depth First Search* (DFS)

Algoritma *Depth First Search* merupakan algoritma untuk menyelesaikan persoalan dengan melakukan pencarian solusi. Pencarian solusi dilakukan dengan pembentukan graf dinamis secara mendalam. Kemudian, dilakukan pengecekkan untuk setiap simpul apakah simpul tersebut merupakan solusinya atau bukan. Jika mencapai kedalaman akhir, tetapi belum merupakan simpul solusi, pencarian akan melakukan *backtrack* ke simpul atasnya, kemudian melakukan pencarian pada simpul dibawahnya selain yang sudah dilewati.

Contoh DFS untuk pembantukan ruang status:



Gambar 2.3.2 Pohon Pembentukan Ruang Satus dengan DFS

Depth First Search memiliki aspek pencarian sebagai berikut:

1. *Completeness,* menjamin ditemukannya solusi jika pencabangan dan kedalaman terbatas.
2. *Optimality,* tidak dapat menjamin solusi optimal.
3. *Time Complexity,* waktu yang dibutuhkan sangat tergantung dengan jumlah maksimal kedalaman suatu graf.
4. *Space Complexity,* lebih baik dalam aspek kompleksitas ruang dibandingkan *Breadth First Search*.

2.5 C# *Desktop Application Development*

Desktop Application merupakan suatu aplikasi atau *software* milik desktop (PC atau laptop). Untuk pengembangan *Desktop Application* dibutuhkan IDE (*Integrated Development Environment*) yang sudah terpasang secara lokal(dapat di-*install* secara online). Untuk pengembangan Desktop Application dengan menggunakan bahasa pemrograman C# dibutuhkan *framework*, seperti .Net.

BAB III
ANALISIS PEMECAHAN MASALAH

3.1 Langkah-langkah pemecahan masalah

3.1.1 Breadth First Search

Mencari satu solusi :

1. Sediakan queue kosong.
2. Masukkan node akar ke dalam *queue.*
3. *Enqueue* pada *queue* (jadikan *current node*). Periksa apakah *current node* tersebut merupakan solusi atau tidak.
   a. Jika node tersebut adalah solusi, pencarian selesai dan umumkan hasil.
   b. Jika node tersebut bukan solusi, lanjutkan pencarian.
4. Periksa apakah *current node* memiliki *child node* atau tidak.
   a. Jika ya, masukkan node anak dari *current node* ke dalam *queue.*
   b. Jika tidak dan seluruh node sudah dikunjungi(*queue* kosong), pencarian selesai dan umumkan hasil tidak ditemukan.
   c. Jika tidak dan masih ada node yang belum dikunjungi (*queue* masih berisi), lanjutkan pencarian.
5. Kembali ke nomor 3.

Mencari semua kejadian :

1. Sediakan queue kosong.
2. Masukkan node akar ke dalam *queue.*
3. *Enqueue* pada *queue* (jadikan *current node*). Periksa apakah *current node* tersebut merupakan solusi atau tidak.
   a. Jika node tersebut adalah solusi, simpan solusi dan lanjutkan pencarian.
   b. Jika node tersebut bukan solusi, lanjutkan pencarian.
4. Periksa apakah *current node* memiliki *child node* atau tidak.
   a. Jika ya, masukkan node anak dari *current node* ke dalam *queue.*
   b. Jika tidak dan seluruh node sudah dikunjungi(*queue* kosong), pencarian selesai dan umumkan hasil jika ada.
   c. Jika tidak dan masih ada node yang belum dikunjungi (*queue* masih berisi), lanjutkan pencarian
5. Kembali ke nomor 3.

3.1.2 Depth First Search

Mencari satu solusi

1. Sediakan *stack* kosong.
2. Masukkan node akar ke dalam *stack.*

3. Ambil node dari *top stack* (jadikan *current node*). Periksa apakah *current node* tersebut merupakan solusi atau tidak.
    a. Jika node tersebut adalah solusi, pencarian selesai dan umumkan hasil.
    b. Jika node tersebut bukan solusi, lanjutkan pencarian.
4. Periksa apakah *current node* memiliki *child node* atau tidak.
    a. Jika ya, masukkan node anak dari *current node* ke dalam *stack.*
    b. Jika tidak dan seluruh node sudah dikunjungi(*stack* kosong), pencarian selesai dan umumkan hasil tidak ditemukan.
    c. Jika tidak dan masih ada node yang belum dikunjungi (*stack* masih berisi), lanjutkan pencarian.
5. Kembali ke nomor 3.

Mencari semua kejadian

1. Sediakan *stack* kosong.
2. Masukkan node akar ke dalam *stack.*
3. Ambil node dari *top stack* (jadikan *current node*). Periksa apakah *current node* tersebut merupakan solusi atau tidak.
    a. Jika node tersebut adalah solusi, simpan solusi..
    b. Jika node tersebut bukan solusi, lanjutkan pencarian.
4. Periksa apakah *current node* memiliki *child node* atau tidak.
    a. Jika ya, masukkan node anak dari *current node* ke dalam *stack.*
    b. Jika tidak dan seluruh node sudah dikunjungi(*stack* kosong), pencarian selesai dan umumkan hasil kalau ada.
    c. Jika tidak dan masih ada node yang belum dikunjungi (*stack* masih berisi), lanjutkan pencarian.
5. Kembali ke nomor 3.

3.2 Proses mapping

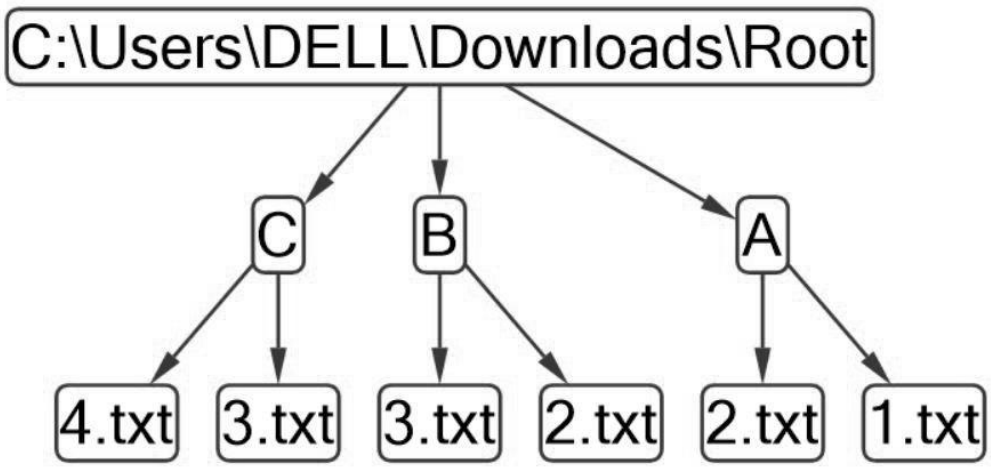
Program menggunakan pustaka class dari C# yaitu Class DirectoryInfo dengan kategori System.IO. Program awalnya menerima dua buah input, yaitu *starting folder* dan *target file.*

Untuk *crawling* menggunakan algoritma *Breadth First Search, starting folder* dijadikan node akar yang dimasukkan ke dalam sebuah queue. Dequeue pada queue dan jadikan *return value*-nya sebagai *current filesystem.* Solusi akan diperiksa sesuai nama *current filesystem.* Apabila solusi belum ditemukan, *current filesystem* yang berjenis folder akan di-list isinya dan di-enqueue ke dalam queue. Kemudian diperiksa kembali dengan seperti menggunakan proses Dequeue sebelumnya.

Untuk *crawling* menggunakan algoritma *Depth First Search, starting folder* dijadikan node akar yang dimasukkan ke dalam sebuah *stack*. *Pop* pada dan jadikan *return value*-nya sebagai *current filesystem.* Solusi akan diperiksa sesuai nama *current filesystem.* Apabila solusi belum ditemukan, *current filesystem* yang berjenis folder akan di-list isinya dan di-*Pop* ke dalam *stack*. Kemudian diperiksa kembali dengan seperti menggunakan proses *Pop* sebelumnya.

3.3 Ilustrasi Kasus

Folder crawling diilustrasikan sebagai tree. Walaupun terdapat node yang memiliki nama sama tetapi node yang menunjuk akan berbeda (nama file dan folder dalam suatu folder adalah unik).



Gambar 3.3.1 Contoh Tree pada Folder

BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi program

4.1.1 Breadth First Search

```
procedure do_bfs(input startingpath : string, input filename: string, input findall:
boolean)

DEKLARASI
   allfolder: List of string  { seluruh folder yang tersedia di bawah startingpath }
   allfile: List of string    { seluruh file yang tersedia di bawah startingpath }
   foundFirst: boolean        { menandakan apakah sudah ketemu first instance dari
                                filename, diinisiasi false di global }
   foundthiscontext: boolean  { digunakan apakah dalam konteks ini, file ditemukan }

ALGORITMA
AddNode(startingpath)
for each file in allfile
    if file = filename and (findall or not foundfirst) then
        foundFirst <-  true
        foundthiscontext <- true
        AddNode(file)
        AddEdge(startingpath, file) { tambahkan edge }
        ColorNode(file, "BLUE")     { warnai node ini sebagai biru }
        UpdateLabel(file)           { update label yang menampilkan clickable path dari
                                      file tersebut di GUI }
        if not findall then
            break                   { sudah ketemu, berarti cukup loopingnya }
    else if (findall or not foundfirst) then { bukan node yang kita cari}
        AddNode(file)
        AddEdge(startingpath, file) { tambahkan edge }
        ColorNode(file, "RED")      { diwarnai merah }
    else
        AddNode(file)               {tidak perlu diwarnai}
        AddEdge(startingpath, file) {tambahkan edge}

    if (findall or not foundfirst) then
      for each folder in allfolder:
        AddNode(folder)
        AddEdge(startingpath, folder)
        do_bfs(startingpath+folder, filename, findall) { lakukan bfs di folder tersebut }

    if (foundthiscontext) then      { menemukan file di konteks ini }
        BacktrackColorEdgeBlue()    { warnai path ke konteks ini dengan warna biru }
```

```
    else
        BacktrackColorEdgeRed()     { warnai path ke konteks ini dengan warna merah }
```

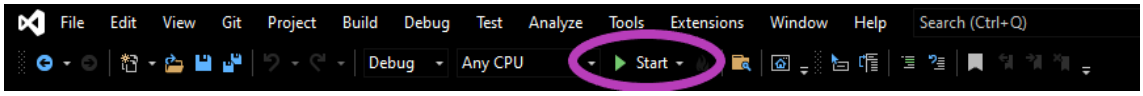
4.1.2 Depth First Search

```
procedure do_bfs(input startingpath : string, input filename: string, input findall:
boolean)

DEKLARASI
   allfolder: List of string  { seluruh folder yang tersedia di bawah startingpath }
   allfile: List of string    { seluruh file yang tersedia di bawah startingpath }
   foundFirst: boolean        { menandakan apakah sudah ketemu first instance dari
                                filename, diinisiasi false di global }
   foundthiscontext: boolean  { digunakan apakah dalam konteks ini, file ditemukan }

ALGORITMA
    AddNode(startingpath)
    for each folder in allfolder:
       AddNode(folder)
       AddEdge(startingpath, folder)    { tambahkan edge }
       do_dfs(startingpath+folder, filename, findall) { lakukan dfs di folder tersebut }

    for each file in allfile:
        if file = filename and (findall or not foundfirst) then
            foundFirst <- true
            foundthiscontext <- true
            AddNode(file)
            AddEdge(startingpath, file) { tambahkan edge }
            ColorNode(file, "BLUE")     { warnai node ini sebagai biru }
            UpdateLabel(file)           { update label yang menampilkan clickable path dari file
                                          tersebut di GUI }
            if not findall then
                break                   { sudah ketemu, berarti cukup loopingnya }
        else if (findall or not foundfirst) then { bukan node yang kita cari}
            AddNode(file)
            AddEdge(startingpath, file) { tambahkan edge }
            ColorNode(file, "RED")      {diwarnai merah}
        else
            AddNode(file) {tidak perlu diwarnai}
            AddEdge(startingpath, file) {tambahkan edge}

    if (foundthiscontext) then          {menemukan file di konteks ini}
        BacktrackColorEdgeBlue()        { warnai path ke konteks ini dengan warna biru }
    else
        BacktrackColorEdgeRed()         { warnai path ke konteks ini dengan warna merah }
```

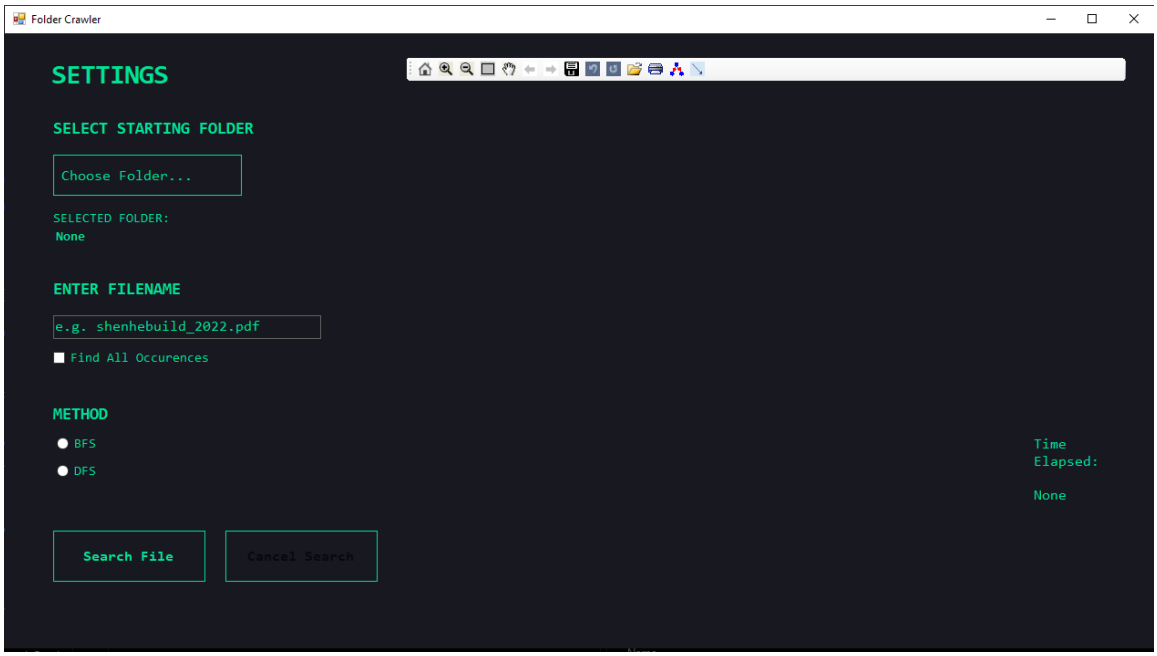
4.2 Struktur Data

4.3 Tata Cara Penggunaan Program

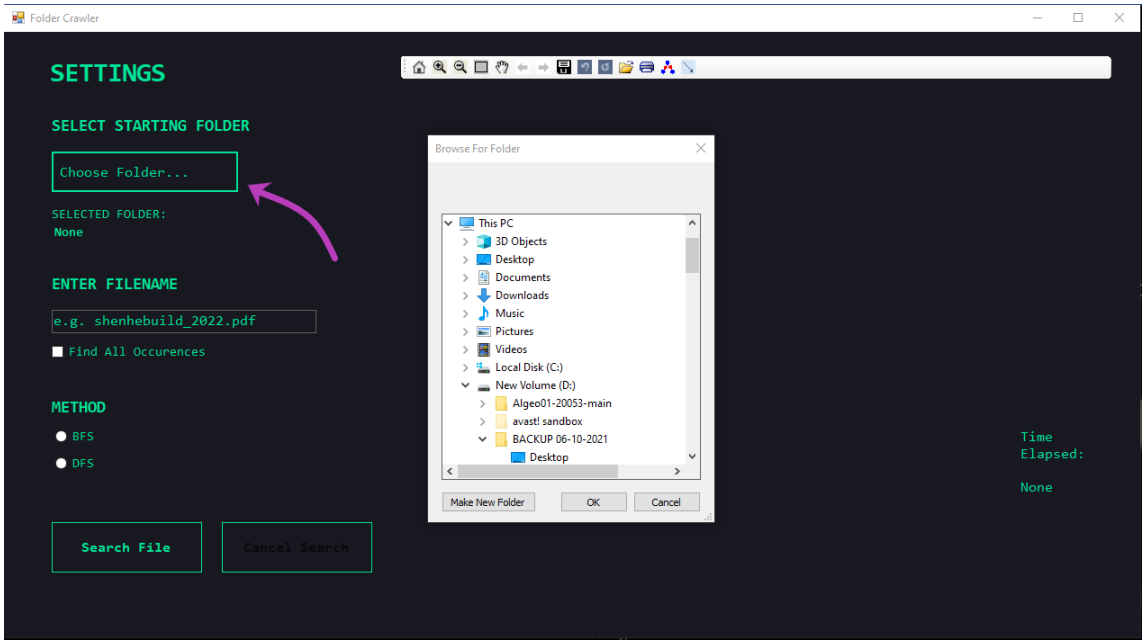
1. Run/Start Program



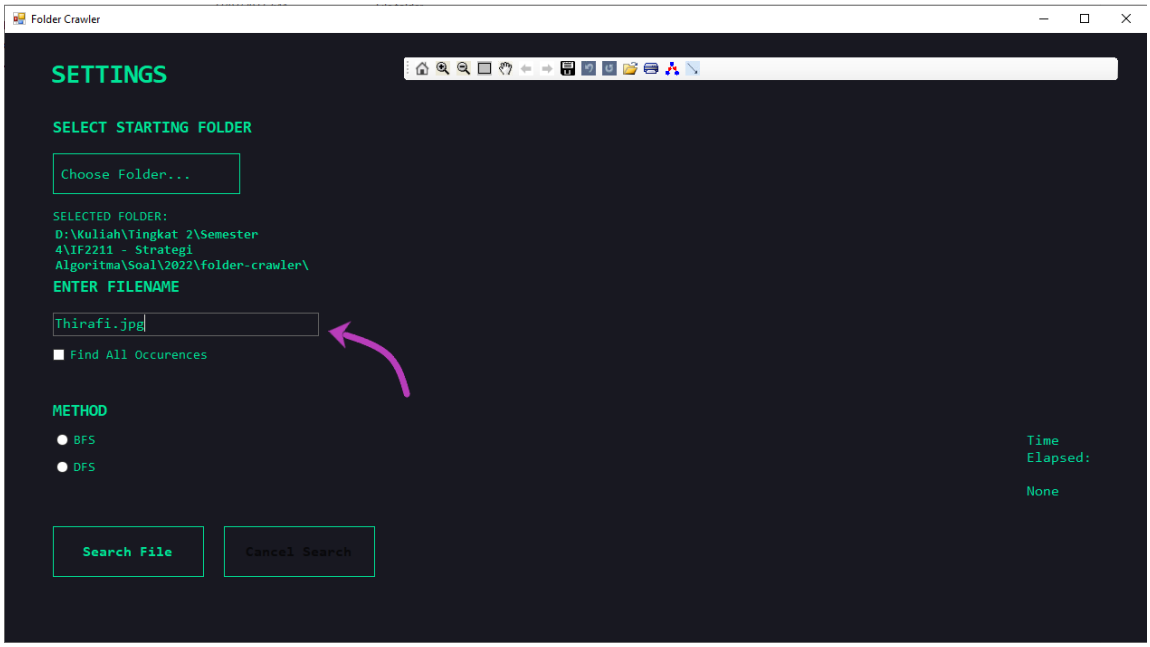
Tampilan aplikasi muncul



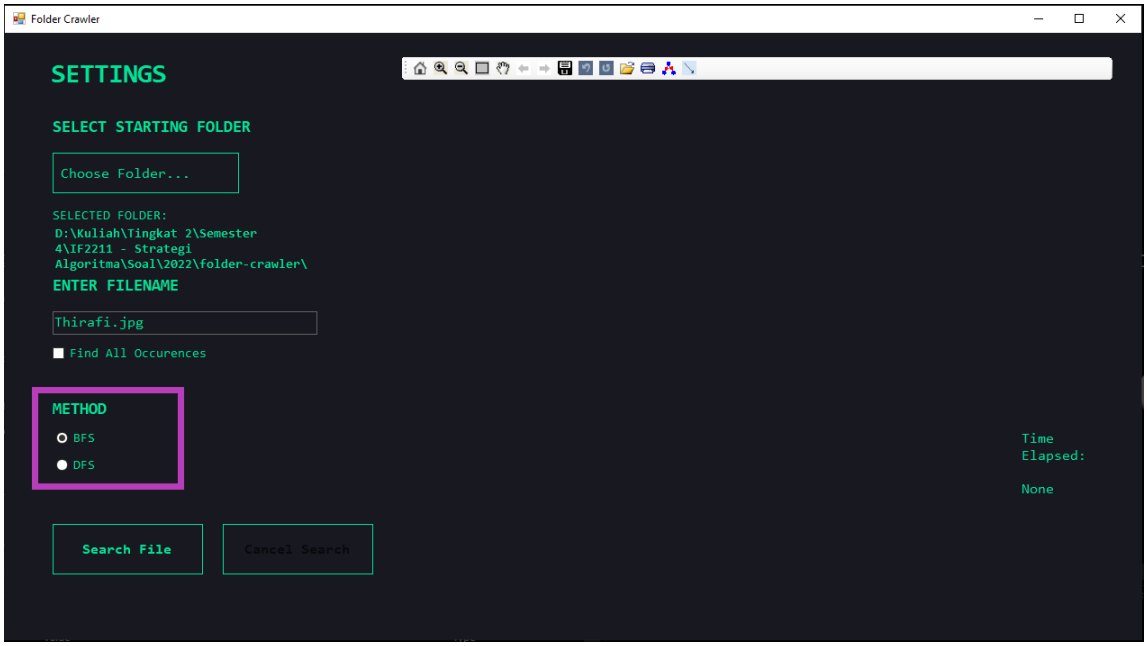
2. Tekan tombol “Choose Folder” dan Pilih Starting Folder



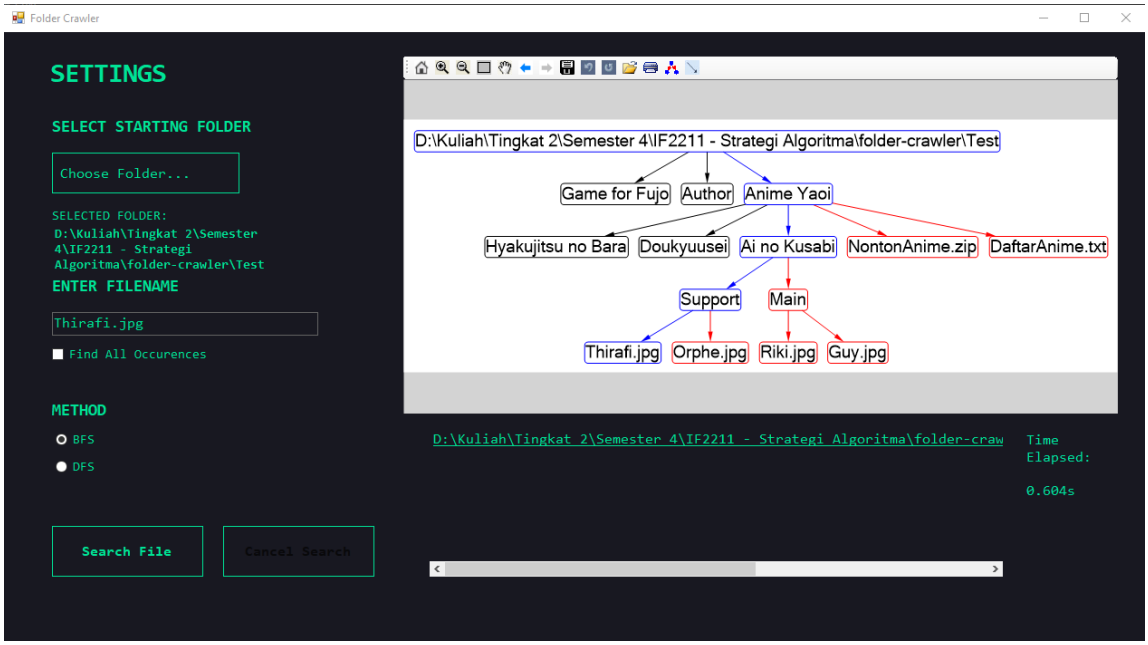
3. Masukkan *filename* yang ingin dicari. Dan dapat diisi checkbox "Find All Occurence"(opsional).



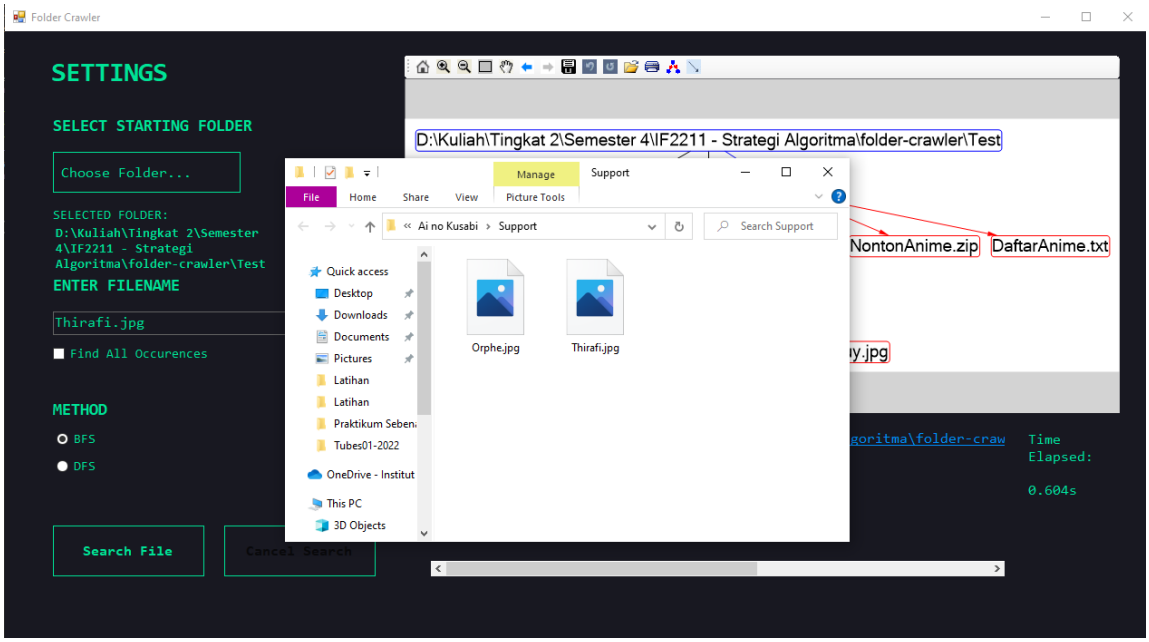
4. Pilih salah satu method (BFS atau DFS).



5. Klik search file untuk melakukan pencarian. Dapat dilakukan *cancel search* pada saat pencarian sedang dilakukan.
   Tampilan setelah search dilakukan



6. User dapat membuka file explorer melalui hyperlink yang ditampilkan

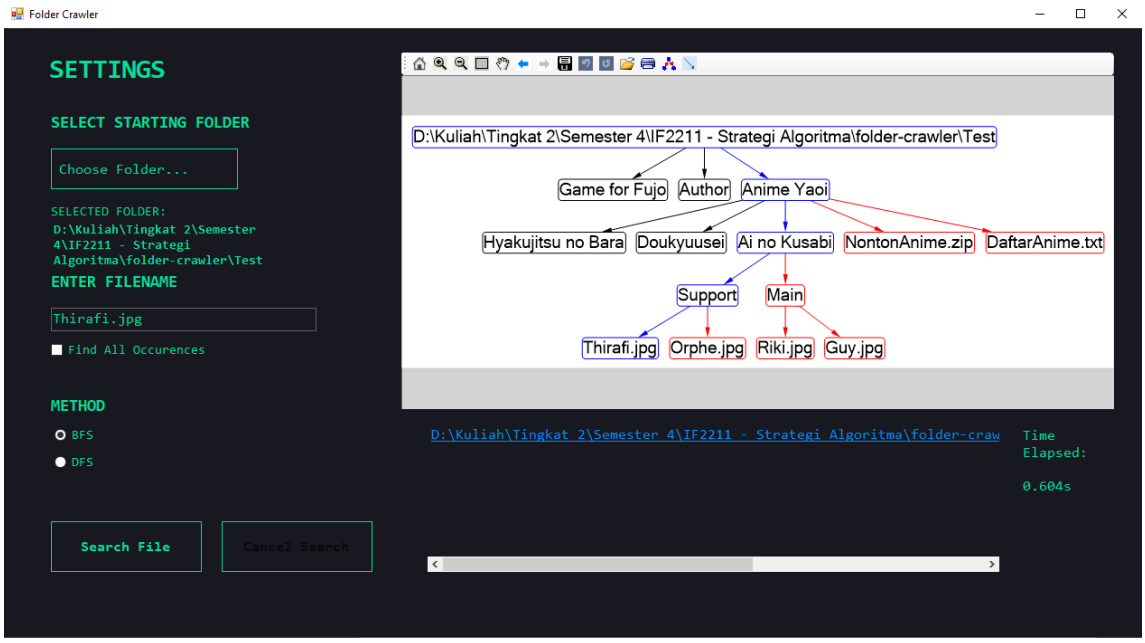


4.4 Hasil pengujian

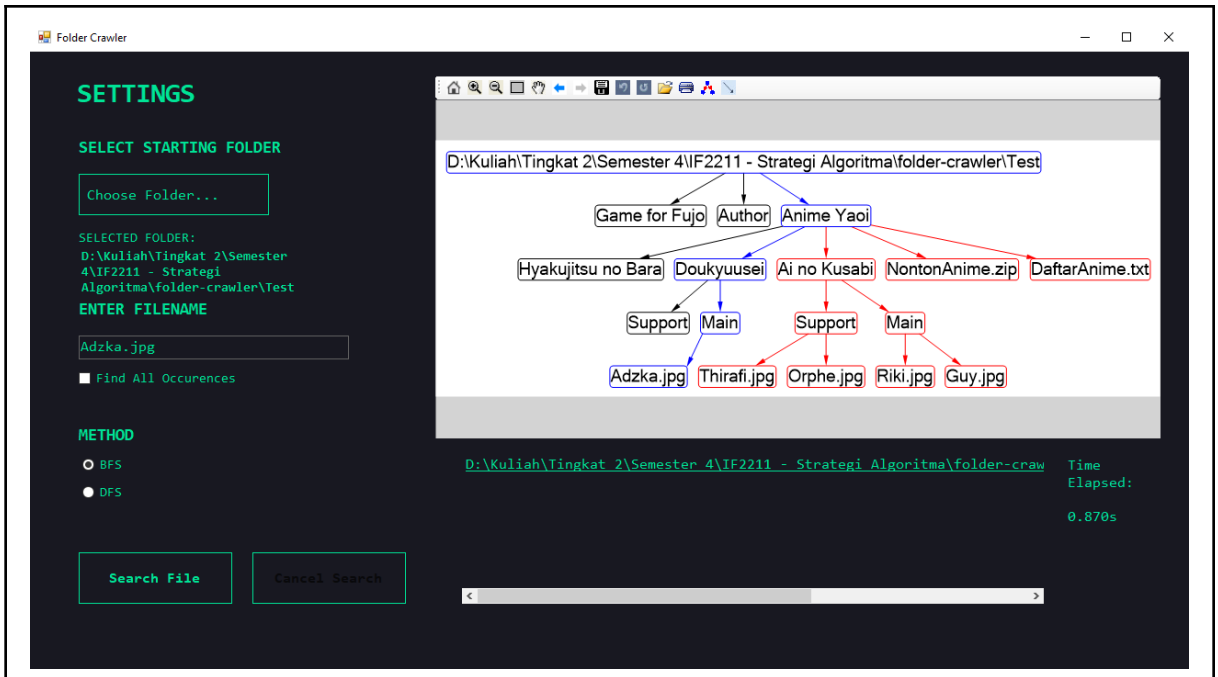
Screenshot di bawah merupakan hasil pengujian untuk beberapa skenario dan setiap penggunaan algoritma BFS dan DFS pada *folder-crawling*. Untuk fitur *find all occurance* akan memberikan hasil yang sama untuk kedua algoritma

4.4.1 Mencari satu solusi

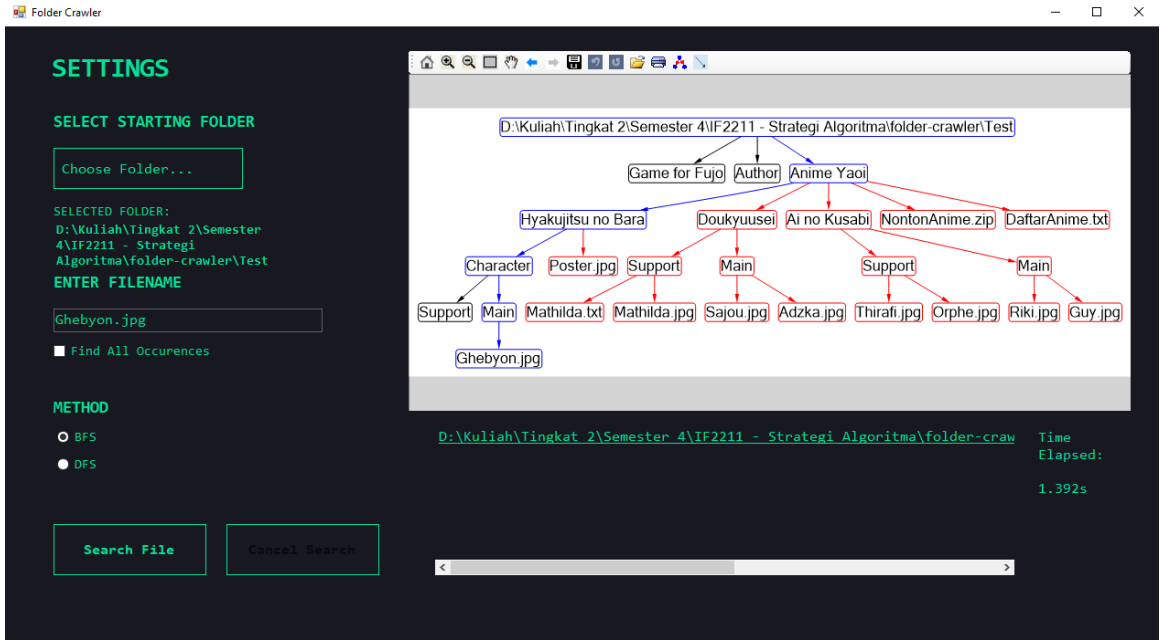
- BFS



Gambar 4.4.1.1 Pencarian Satu File dengan Algoritma BFS(1)

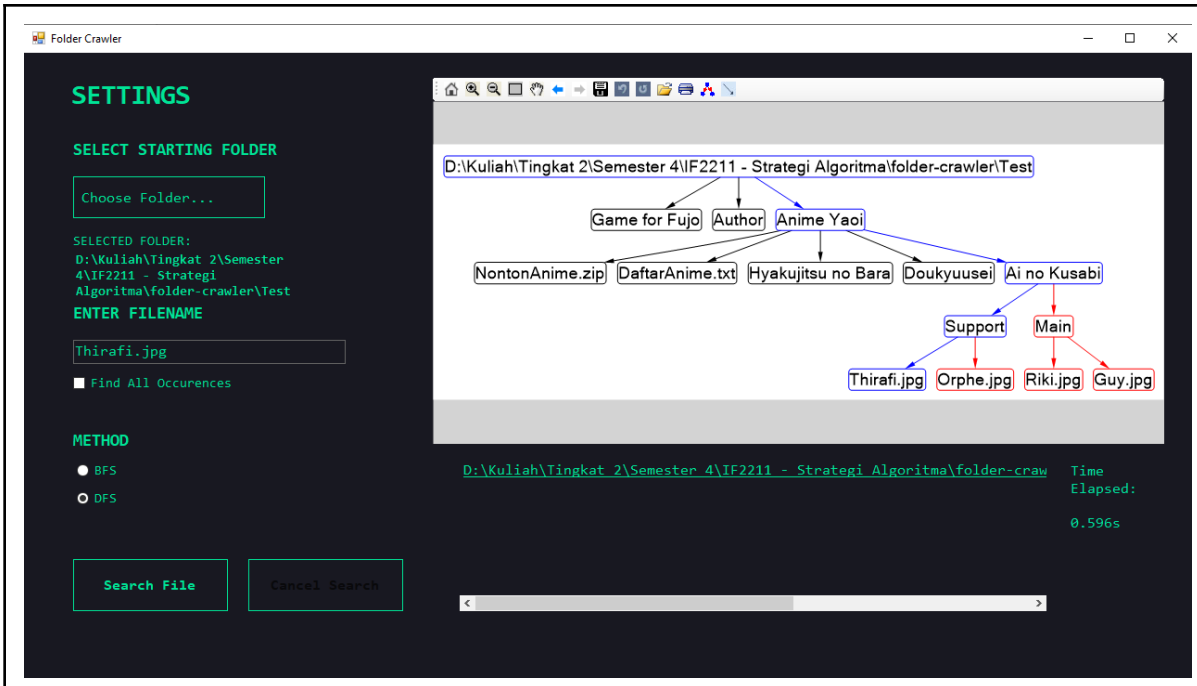


Gambar 4.4.1.2 Pencarian Satu File dengan Algoritma BFS(2)

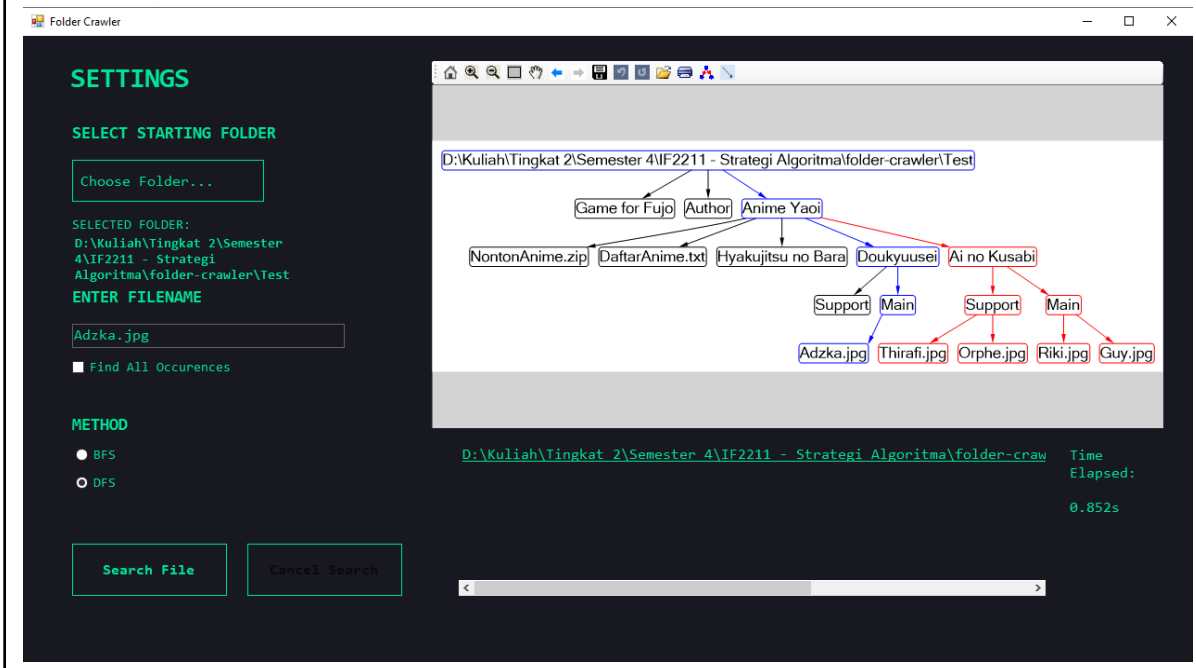


Gambar 4.4.1.3 Pencarian Satu File dengan Algoritma BFS(3)

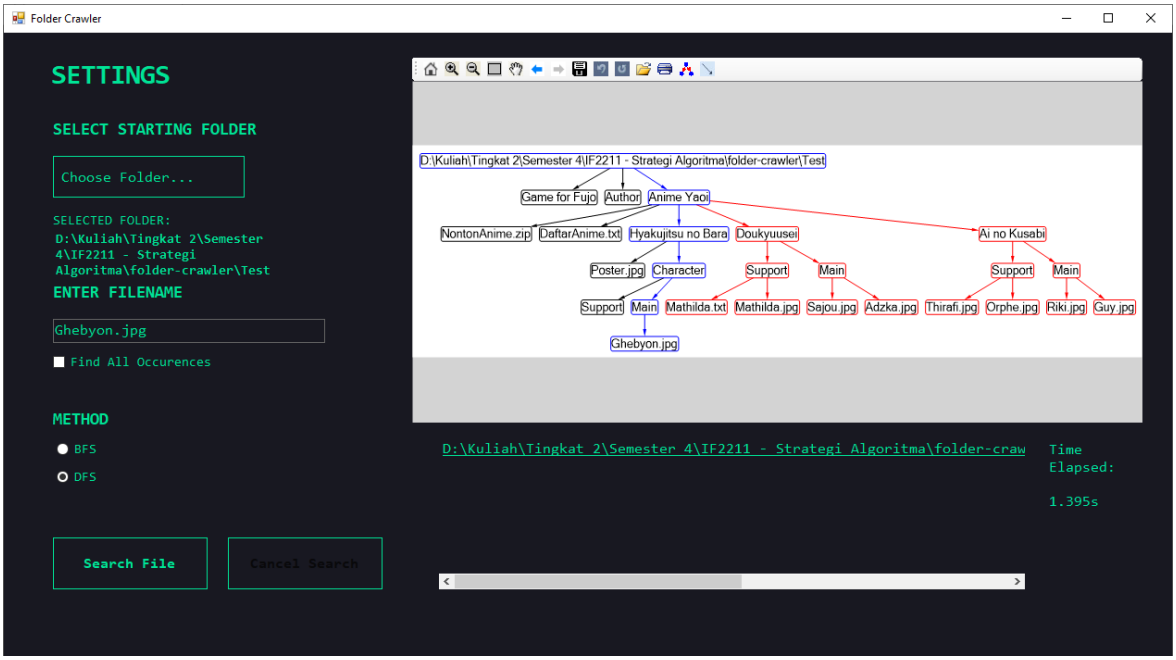
- DFS



Gambar 4.4.1. Pencarian Satu File dengan Algoritma DFS(1)

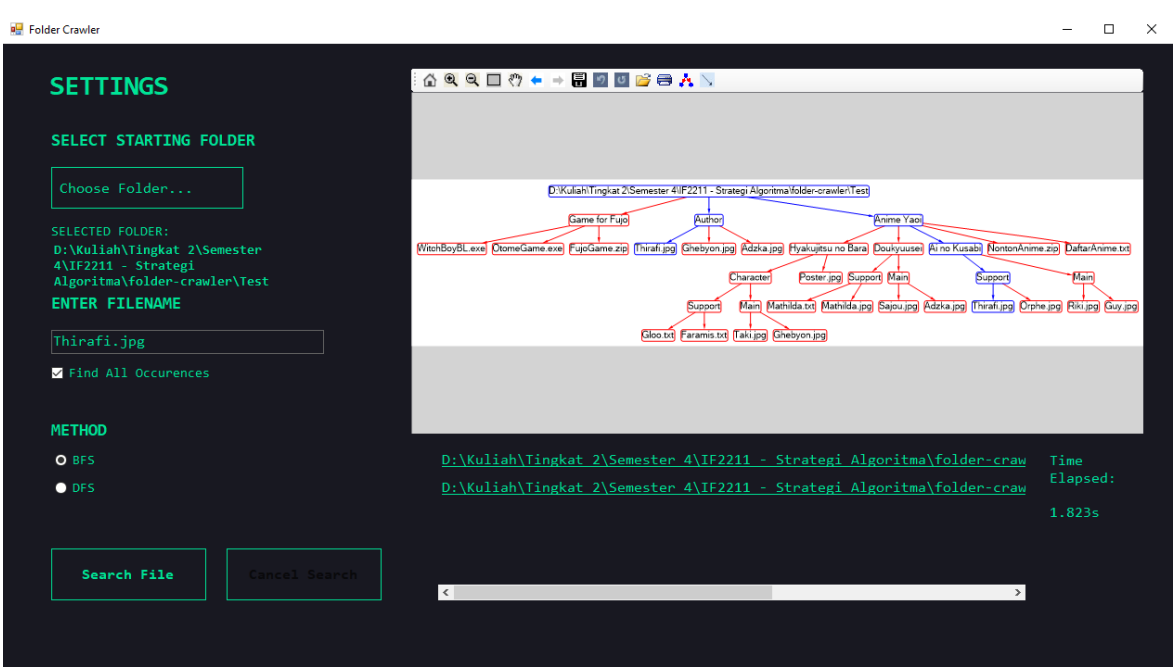


Gambar 4.4.1.5 Pencarian Satu File dengan Algoritma DFS(2)

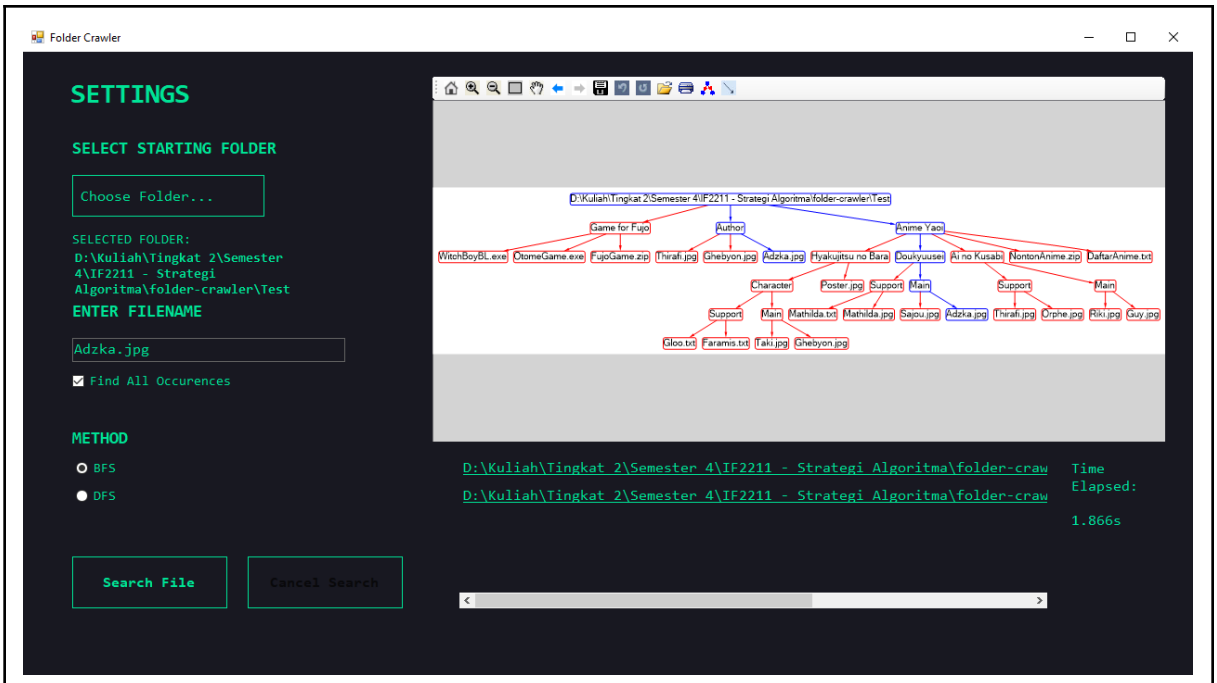


Gambar 4.4.1.6 Pencarian Satu File dengan Algoritma DFS(3)

4.4.2 *Find All Occurance*



Gambar 4.4.2.1 Pencarian File pada Seluruh Kejadian (1)



Gambar 4.4.2.2 Pencarian File pada Seluruh Kejadian (2)

Gambar 4.4.2.3 Pencarian File pada Seluruh Kejadian (3)

4.5 Analisis Desain Solusi Algoritma BFS dan DFS

Secara umum, kedua strategi memiliki performa dan pencarian waktu yang tidak berbeda jauh. Terlebih ketika dilakukan pencarian seluruh kemunculan solusi, maka kedua strategi akan

melakukan pembandingan yang sama karena seluruh simpul kemungkinan solusi akan ditelusuri walau sudah ditemui satu atau lebih solusi. Secara teori, BFS akan lebih cepat menemukan solusinya jika solusi tersebut ada di kedalaman yang relatif dekat dengan simpul akar. Sedangkan DFS akan lebih cepat ketika solusi terdapat pada kedalaman yang jauh dari simpul akar dibanding BFS.

Strategi DFS lebih baik dari strategi BFS ketika solusi yang dicari berada pada kedalaman yang relatif jauh dari simpul akar. Hal tersebut dikarenakan strategi BFS akan melakukan pencarian seluruh kedalaman di atas solusi terlebih dahulu. Oleh karena itu, jika terdapat banyak pencabangan, BFS akan memakan memory dan ruang yang cukup banyak.

Bab 5
Kesimpulan dan Saran

5.1 Kesimpulan

Strategi BFS dan DFS dapat diimplementasikan di pencarian *file* dari suatu simpul akar berupa *folder* dengan menggunakan C# *Desktop Application Development* serta GUI dan visualisasi graf pohon untuk menunjukkan proses pencarian dengan strategi tersebut. Kedua strategi memiliki performa dan memakan waktu yang mirip. Namun, terdapat beberapa kasus dimana DFS lebih baik dibanding BFS, dan BFS lebih baik dibanding DFS. Selain itu, ketika terdapat pencabangan yang sangat banyak, BFS dapat memakan memori dan ruang yang lebih banyak dibanding DFS.

5.2 Saran

Berikut beberapa saran yang diberikan oleh pembuat program tugas besar ini yang mungkin dapat berguna bagi siapapun yang akan mengerjakan tugas serupa:

1. Pahami terlebih dahulu teori dan konsep dasar terkait algoritma dan pelajari pseudo-code setiap algoritma.
2. Sebelum memulai pengimplementasian algoritma kedalam kode C#, pahami dan buat bagian GUI-nya terlebih dahulu serta pahami penggunaan MSAGL untuk visualisasi grafnya.
3. Lakukan dekomposisi permasalahan apa saja yang perlu diimplementasikan kedalam program terlebih dahulu hingga lebih mudah untuk dibuat solusinya karena langsung memikirkan penyelesaian secara keseluruhan akan terasa sangat sulit dan membuat kewalahan.

Link Repository dan Video

- Repository GitHub : https://github.com/reverseon/folder-crawler
- Video : https://youtu.be/GW7voWfN4V4

Daftar Pustaka

1) “Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1)”. informatika.stei.itb.ac.id. 2022. Diakses pada tanggal 22 Maret 2022. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf
2) “Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 2)”. informatika.stei.itb.ac.id. 2022. Diakses pada tanggal 22 Maret 2022. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf
3) “Tugas besar 2: Pengaplikasian Algoritma BFS dan DFS dalam Implementasi Folder Crawling”. 2022. https://cdn-edunex.itb.ac.id/38015-Algorithm-Strategies-Parallel-Class/85259-BFS-dan-DFS/1646201812962_Tugas-Besar-2-IF2211-Strategi-Algoritma-2022.pdf