

LAPORAN TUGAS BESAR

IF2211 STRATEGI ALGORITMA

Pengaplikasian Algoritma BFS dan DFS dalam Implementasi *Folder-Crawling*



Dafeb's Fanclub

Disusun oleh :

Kelompok 13

Rayhan Kinan Muhamnad	13520065
Brianaldo Phandiarta	13520113
David Karel Halomoan	13520154

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2022

DAFTAR ISI

DAFTAR ISI.....	2
BAB I DESKRIPSI TUGAS.....	3
BAB II LANDASAN TEORI	9
2.1. Dasar Teori.....	9
2.2. C# Desktop Application Development	12
BAB III ANALISIS PEMECAHAN MASALAH	14
3.1. Langkah-langkah Pemecahan Masalah	14
3.2. Elemen-elemen Algoritma BFS dan DFS.....	14
3.3. Ilustrasi Kasus Lain.....	15
BAB IV IMPLEMENTASI DAN PENGUJIAN	18
4.1. Implementasi Program	18
4.2. Struktur Data dan Spesifikasi Program.....	21
4.3. Tata Cara Penggunaan Program.....	23
4.4. Hasil Pengujian	24
4.5. Analisis Desain Solusi Algoritma BFS dan DFS.....	31
BAB V KESIMPULAN DAN SARAN	32
5.1. KESIMPULAN.....	32
5.2. SARAN	32
DAFTAR PUSTAKA.....	33

BAB I

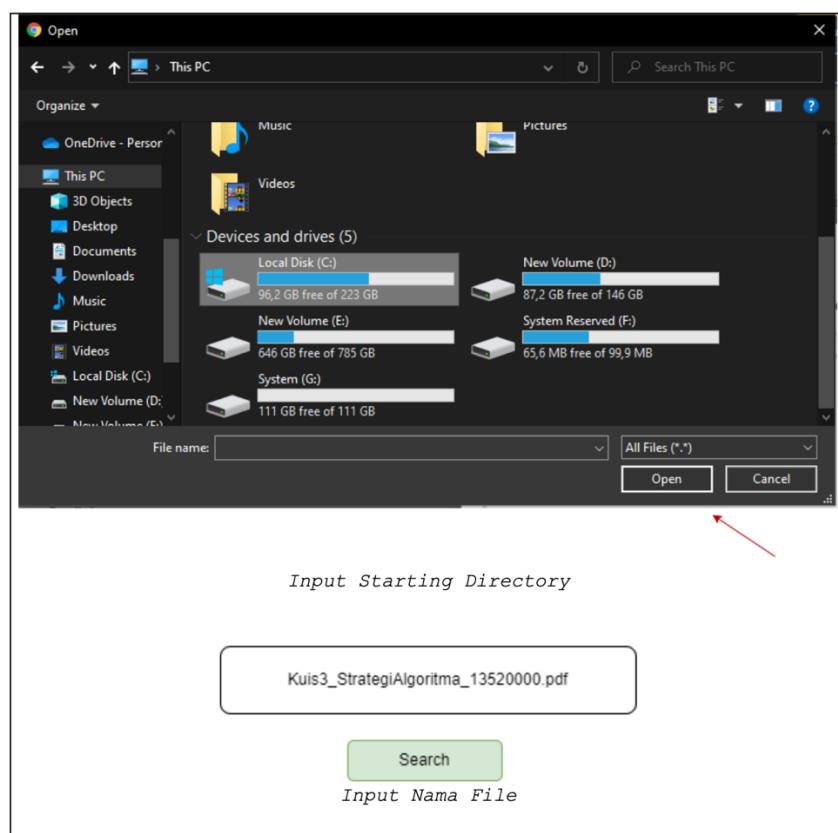
DESKRIPSI TUGAS

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. Contoh hal-hal yang dimaksud akan dijelaskan di bawah ini.

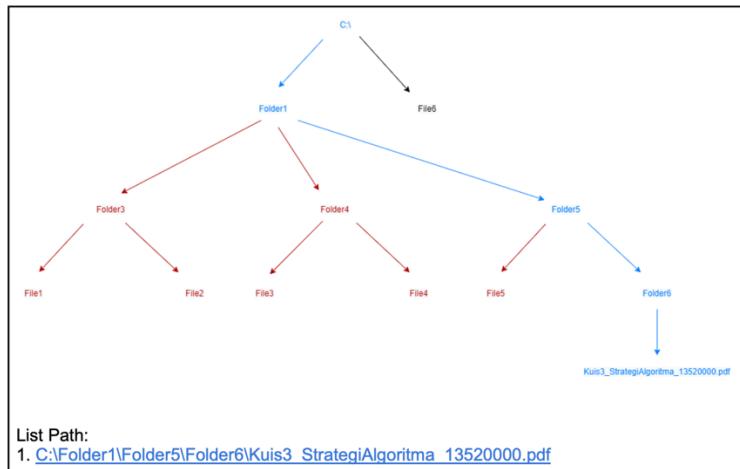
Contoh Input dan Output Program

Contoh masukan aplikasi:



Gambar 1.1. Contoh Input Program

Contoh output aplikasi:

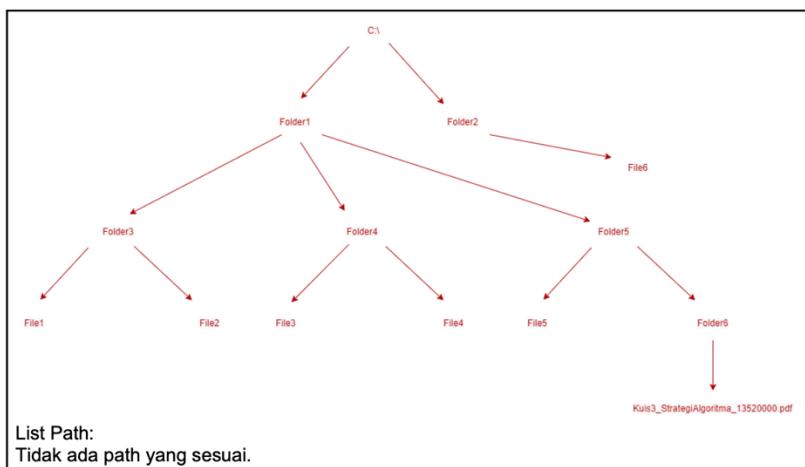


Gambar 1.2. Contoh output program

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.

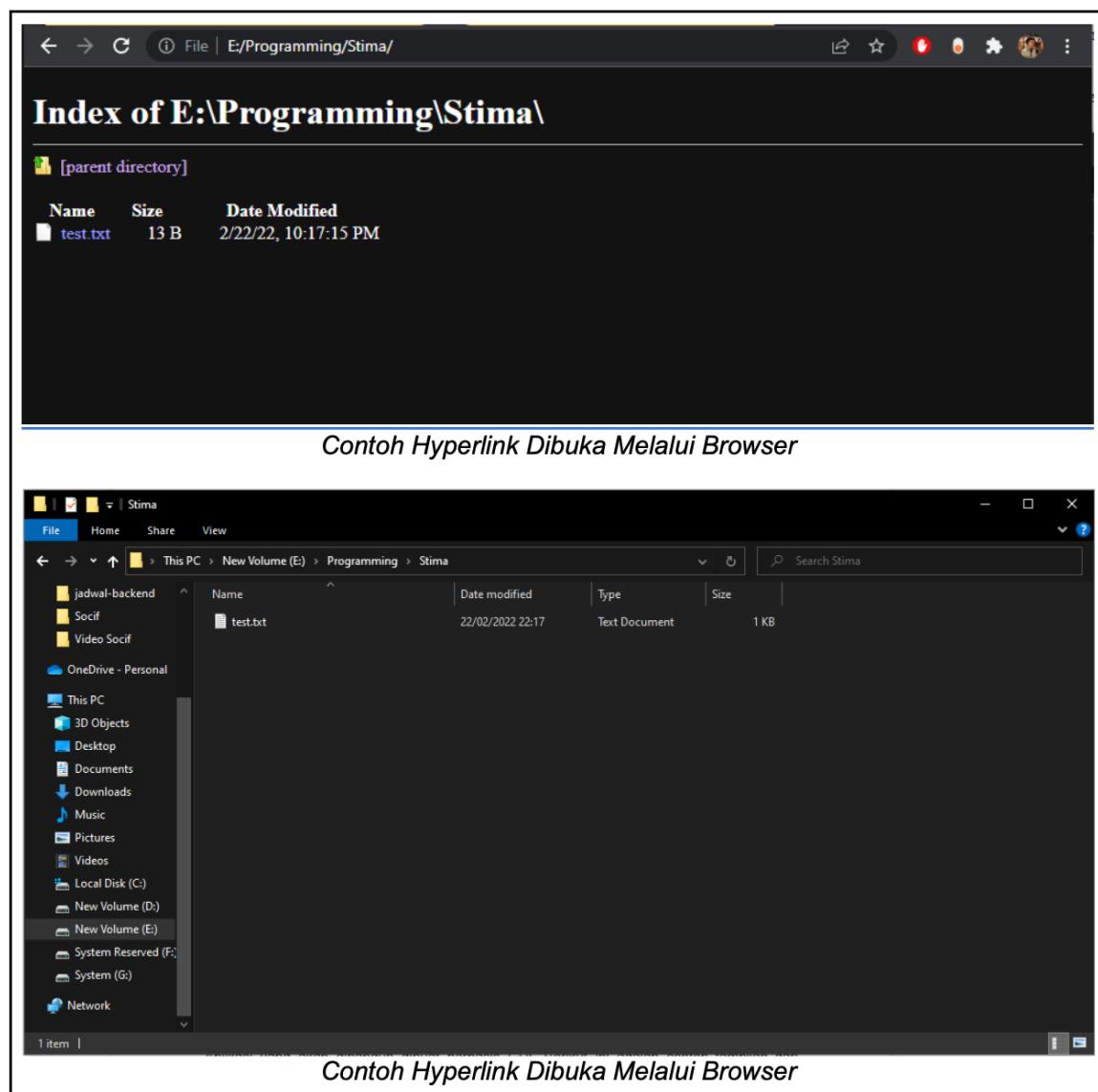


Gambar 1.3. Contoh output program jika file tidak ditemukan

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 1.4. Contoh ketika hyperlink di-klik

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

Folder Crawling

<p>Input</p> <p>Choose Starting Directory <input type="button" value="Choose Folder.."/> No File Chosen</p> <p>Input File Name <input type="text" value="e.g. \" word.pdf\""=""/> <input type="checkbox"/> Find all occurrence</p> <p>Input Metode Pencarian <input type="radio"/> BFS <input checked="" type="radio"/> DFS</p> <p><input type="button" value="Search"/></p>	<p>Output</p> <div style="background-color: #cccccc; height: 150px; width: 100%;"></div>
---	---

Folder Crawling

<p>Input</p> <p>Choose Starting Directory <input type="button" value="Change Folder.."/> C:/</p> <p>Input File Name <input type="text" value="Kuis3_StrategiAlgoritma_13520000.pdf"/> <input type="checkbox"/> Find all occurrence</p> <p>Input Metode Pencarian <input type="radio"/> BFS <input checked="" type="radio"/> DFS</p> <p><input type="button" value="Search"/></p>	<p>Output</p> <div style="border: 1px solid black; padding: 10px; width: 100%;"><pre>graph TD; C1[C:/] --> F1[Folder1]; C1 --> F2[Folder2]; C1 --> F6[File6]; F1 --> F1_1[File1]; F1 --> F1_2[File2]; F1 --> F3[Folder3]; F3 --> F3_1[File3]; F3 --> F3_2[File4]; F2 --> F5[File5]; F2 --> F5_1[Folder5]; F5_1 --> F6_1[Folder6]</pre><p>Kuis3_StrategiAlgoritma_13520000.pdf</p></div> <p>Path File : • C:/Folder1/Folder5/Folder6/Kuis3_StrategiAlgoritma_13520000.pdf</p> <p>Time spent: 20.02s</p>
---	--

Gambar 1.5. 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 parentnya.

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. Dasar Teori

Graf traversal atau algoritma traversal dalam graf adalah algoritma pengunjungan simpul-simpul dengan cara yang sistematik. Algoritma traversal dalam graf terdiri dari dua, yaitu pencarian melebar (*Breadth-First Search/BFS*) dan pencarian mendalam (*Depth-First Search/DFS*).

Breadth-First Search atau BFS merupakan salah satu algoritma yang paling sederhana dalam melakukan pencarian pada graf dan pola dasar pada algoritma pencarian graf lainnya. Algoritma *minimum-spanning-tree* Prim dan Algoritma pencarian jarak terdekat Dijkstra menggunakan ide yang serupa dengan pencarian *breadth-first search*. Dalam algoritma *breadth-first search*, graf direpresentasikan dalam bentuk $G = (V, E)$ dengan v merupakan simpul awal penulusuran.

Secara sederhana, algoritma *breadth-first search* akan memulai penulusuran dari simpul v . Kemudian, akan dilakukan penelusuran terhadap semua simpul yang bertetangga dengan simpul v terlebih dahulu. Terakhir, akan dilakukan penelusuran terhadap simpul-simpul lain yang belum dikunjungi yang bertetangga dengan simpul-simpul yang telah dikunjungi. Langkah ini akan dilakukan terus-menerus hingga ditemukan simpul yang dicari atau hingga seluruh simpul telah ditelusuri.

Pada prosedur *breadth-first search*, dapat digunakan *adjacency lists* dalam merepresentasikan graf $G = (V, E)$. Selain itu, untuk mengetahui simpul yang akan diperiksa, akan digunakan struktur data Queue. Terakhir, untuk mengetahui suatu simpul telah diperiksa atau tidak, akan digunakan struktur data *array* atau *hash table* yang bertipe boolean. Berikut adalah *pseudocode* umum untuk prosedur *breadth-first search*.

```

procedure BFS(input G: graph, input v: vertice)
{ Traversal Graf dengan algoritma penelusuran BFS
  I.S. G dan v terdefinisi dan tidak sembarang
  F.S. Semua simpul yang dilalui tercetak pada layar }

KAMUS
{ Variabel }
q : queue
w : vertice
visited : hashTable
{ Fungsi dan Prosedur antara }
procedure createQueue(input/output q: queue)
{ Inisialisasi queue
  I.S. q belum terdefinisi
  F.S. q terdefinisi dan kosong }
procedure enqueue(input/output q: queue, input v: vertice)
{ Memasukkan v ke dalam q dengan aturan FIFO
  I.S. q terdefinisi
  F.S. v masuk ke dalam q dengan aturan FIFO }
function dequeue(q: queue) → vertice
{ Menghapus simpul dari q dengan aturan FIFO
  dan mengembalikan simpul yang dihapus}
function isQueueEmpty(q: queue) → boolean
{ Mengembalikan True jika q kosong dan sebaliknya }
procedure createHashTable(input/output T: hashTable)
{ Inisialisasi hashTable
  I.S. T belum terdefinisi
  F.S. T terdefinisi dan terisi False sebanyak simpul }
procedure setHashTable(input/output T: hashTable, input v: vertice, input val: boolean)
{ Mengubah value dari T
  I.S. T sudah terdefinisi
  F.S. Elemen T dengan key v sudah diubah menjadi val }
function getHashTable(T: hashTable, input v: vertice) → boolean
{ Mengembalikan elemen T pada key v }

ALGORITMA
{ Inisialisasi visited }
createHashTable(visited)
{ Inisialisasi q }
createQueue(q)

{ Mengunjungi simpul pertama (akar) }
output(v)
enqueue(q, v)
setHashTable(T, v, True

{ Mengunjungi semua simpul }
while (not isQueueEmpty(q)) do
  v ← dequeue(q)
  for setiap simpul w yang bertetangga dengan simpul v do
    if (not getHashTable(visited, w)) then
      output(w)
      enqueue(q, w)
      setHashTable(T, w, True

{ q kosong }

```

Pseudocode umum untuk prosedur *breadth-first search* di atas akan menghasilkan seluruh simpul yang ada. Untuk pencarian satu simpul, algoritma dapat ditambahkan menjadi seperti berikut.

```

procedure BFS(input G: graph, input v: vertice, input x: vertice)
{ Traversal Graf dengan algoritma penelusuran BFS
  I.S. G dan v terdefinisi dan tidak sembarang
  F.S. Semua simpul yang dilalui tercetak pada layar }

KAMUS
{ Variabel }
{ IDEM }
found : boolean
{ Fungsi dan Prosedur antara }
{ IDEM }

ALGORITMA
{ Inisialisasi found }
found ← False
{ Inisialisasi visited }
createHashTable(visited)
{ Inisialisasi q }
createQueue(q)

{ Mengunjungi simpul pertama (akar) }
output(v)
enqueue(q, v)
setHashTable(T, v, True

{ Memeriksa simpul pertama (akar) }
if (v = x) then
  found ← True

{ Mengunjungi semua simpul }
while (not isEmpty(q) and not found) do
  v ← dequeue(q)
  for setiap simpul w yang bertetangga dengan simpul v do
    if (not getHashTable(visited, w) and not found) then
      output(w)
      enqueue(q, w)
      setHashTable(T, w, True)
      if (w = x) then
        found ← True

  if (not found) then
    output("Tidak ditemukan")

```

Berbeda dengan *breadth-first search*, algoritma pencarian *depth-first search* melakukan penelusuran terlebih dahulu ke-“dalam” ketika memungkinkan. Algoritma *depth-first search* akan menelusuri simpul tetangga dari simpul yang ditelusuri sekarang. Setelah sudah tidak ada simpul tetangga yang tersedia, algoritma akan melakukan *backtracks* untuk menelusuri simpul-simpul tetangga dari simpul sebelumnya.

Secara sederhana, algoritma *depth-first search* akan memulai penelusuran dari simpul v . Kemudian, akan dilanjutkan penelusuran ke simpul w yang bertetangga dengan simpul v . Setelah itu akan dilakukan pemanggilan kembali algoritma DFS yang dimulai dari simpul w . Ketika mencapai simpul u sedemikian sehingga semua simpul yang bertetangga telah dikunjungi, akan dilakukan pencarian runut-balik (*backtrack*) ke simpul terakhir yang dikunjungi sebelumnya, yaitu simpul yang melakukan pemanggilan algoritma DFS. Pencarian selesai jika tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.

Pada prosedur *depth-first search*, representasi graf dan struktur data hampir serupa seperti yang digunakan pada prosedur *breadth-first search*. Akan tetapi, pada *depth-first search*, tidak akan digunakan struktur data queue, melainkan lebih mirip dengan struktur data stack. Berikut adalah *pseudocode* dari prosedur *depth-first search*.

```

procedure DFS(input G: graph, input/output visited: hashTable, input v: vertice)
{ Traversal Graf dengan algoritma penelusuran DFS
  I.S. G, v, dan visited terdefinisi dan tidak sembarang
  F.S. Semua simpul yang dilalui tercetak pada layar }
KAMUS
{ Variabel }
w : vertice
{ Fungsi dan Prosedur antara }
procedure setHashTable(input/output T: hashTable, input v: vertice, input val: boolean)
{ Mengubah value dari T
  I.S. T sudah terdefinisi
  F.S. Elemen T dengan key v sudah diubah menjadi val }
function getHashTable(T: hashTable, input v: vertice) → boolean
{ Mengembalikan elemen T pada key v }

ALGORITMA
{ Mengunjungi simpul sekarang }
output(v)
setHashTable(T, v, True)

{ Mengunjungi semua simpul }
for setiap simpul w yang bertetangga dengan simpul v do
  if (not getHashTable(visited, w)) then
    DFS(G, visited, w)
  
```

2.2. C# Desktop Application Development

C# Desktop Application Development adalah sebuah kelas pengembang *desktop application* yang mampu beroperasi tanpa menggunakan internet. Pada umumnya, pengembangan *desktop application* menggunakan bahasa pemrogramman C++, C#, ataupun Java. Pengembangan *desktop application*, terlebih dalam hal *user interface* (UI), dipermudah dengan menggunakan WinForms/WPF dan bantuan *integrated development environment* (IDE) Visual Studio. WPF digunakan untuk pengembangan *desktop application* berbasis Windows dengan menggunakan bahasa *native*-nya, yaitu bahasa pemrograman C#.

Berikut merupakan langkah-langkah untuk mengembangkan *desktop application* dengan menggunakan Windows Form App:

1. Membuat *Project* baru:
 - a. Buka aplikasi Visual Studio 2022.
 - b. Pada jendela awal, pilih *Create a new project*.
 - c. Pada jendela *Create a new project*, pilih *Windows Forms App (.NET Framework)*.

- d. Pada jendela *Configure your new project*, isi nama *project* pada isian *Project name*. Kemudian, tekan tombol *Create*.
- 2. Membuat aplikasi:
 - a. Pilih menu *Toolbox* untuk membuka jendela *Toolbox fly-out*.
 - b. Pada jendela *Toolbox fly-out*, kita dapat memilih komponen-komponen yang ingin kita gunakan pada aplikasi desktop kita. Untuk memodifikasi komponen, dapat mengubah properti pada jendela *properties*.
- 3. Menambahkan kode pada *form*:
 - a. Pada jendela Form1.cs [Design], tekan dua kali untuk membuka jendela jendela Form1.cs.
 - b. Kode dapat ditulis pada file Form1.cs.
- 4. Menjalankan aplikasi
 - a. Tekan tombol *Start* pada *menu*.

BAB III

ANALISIS PEMECAHAN MASALAH

3.1. Langkah-langkah Pemecahan Masalah

Langkah awal dalam proses mendesain solusi dari permasalahan ini adalah dengan membagi permasalahan utama, yaitu *folder crawling*, menjadi beberapa permasalahan kecil sehingga dapat mempermudah pencarian solusi. Permasalahan tersebut dapat dibagi menjadi dua permasalahan utama, yaitu pencarian *file* menggunakan algoritma *breadth-first search* dan algoritma *depth-first search*. Selain itu, ada permasalahan-permasalahan tambahan yang perlu untuk diselesaikan yaitu solusi dari dua permasalahan utama tersebut harus ditampilkan dalam bentuk *desktop application*.

Setelah membagi permasalahan menjadi dua, yaitu pencarian menggunakan algoritma *breadth-first search* dan algoritma *depth-first search*, langkah selanjutnya adalah dengan melakukan pemetaan permasalahan ke dalam elemen-elemen algoritma *breadth-first search* dan algoritma *depth-first search*. Setelah melakukan pemetaan elemen-elemen algoritma, dilakukan perancangan algoritma sesuai dengan elemen yang telah dipetakan. Selain itu, kami menambahkan sedikit heuristik pada algoritma dengan cara memeriksa seluruh file terlebih dahulu pada kedalaman sekarang sebelum memeriksa folder.

Dalam menyelesaikan permasalahan tambahan, yaitu menampilkan solusi dari pencarian menggunakan algoritma *breadth-first search* dan *depth-first search*, kami menggunakan bantuan WinForms. Selain itu, kami juga menggunakan bantuan pustaka MSAGL dalam menampilkan visualisasi dari pohon yang telah dibentuk.

3.2. Elemen-elemen Algoritma BFS dan DFS

3.2.1. Breadth-First Search

Algoritma *breadth-first search* pada permasalahan ini menggunakan konsep pencarian solusi dari graf dinamis. Dalam merepresentasikan graf, digunakan daftar ketetanggan atau *adjacency list*. Elemen antrian atau *queue* berisi dengan simpul atau *node* yang akan dikunjungi. Setiap simpul berisi nama *folder* atau *file*. Selain itu, pada permasalahan ini, tidak diperlukan suatu variabel untuk menyimpan simpul yang telah dikunjungi karena permasalahan ini karena direktori pasti memiliki struktur pohon.

3.2.2. Depth-First Search

Algoritma *depth-first search* pada permasalahan ini menggunakan konsep pencarian solusi dari graf dinamis. Dalam merepresentasikan graf, digunakan daftar ketetanggan atau *adjacency list*. Elemen tumpukan atau *stack* berisi dengan simpul atau *node* yang akan dikunjungi. Setiap simpul berisi nama *folder* atau *file*. Selain itu, pada permasalahan ini, tidak diperlukan suatu variabel untuk menyimpan simpul yang telah dikunjungi karena permasalahan ini karena direktori pasti memiliki struktur pohon.

3.3. Ilustrasi Kasus Menggunakan Algoritma Lain

Selain algoritma *breadth-first search* dan algoritma *depth-first search* dalam penelusuran graf, terdapat algoritma lain yang merupakan gabungan dari *breadth-first search* dan *depth-first search*, yaitu algoritma *iterative deepening search* atau IDS. Algoritma ini menggabungkan efisiensi ruang dari algoritma *depth-first search* dan penemuan cepat untuk simpul yang dekat dengan akar dari *breadth-first search*.

Algoritma *iterative deepening search* memanfaatkan algoritma *depth-limited search* atau DLS, yaitu algoritma *depth-first search* dengan pembatasan kedalaman. Berikut merupakan algoritma umum untuk *iterative deepening search* dengan bantuan *depth-first search*.

```
function DLS(src: vertice, target: vertice, limit: integer) → boolean
{ Mengembalikan True jika target ditemukan dan
  False jika target tidak ditemukan dan sudah mencapai kedalaman limit }
KAMUS
{ Variabel }
found : boolean
{ Fungsi dan Prosedur antara }
{ - }

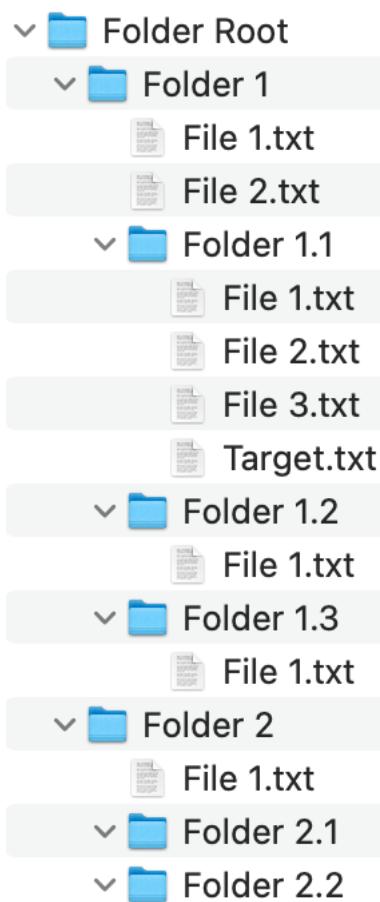
ALGORITMA
if src = target then
  → True
else
  if limit ≤ 0 then
    → False
  else
    found ← false
    for setiap tetangga dari src do
      found ← found or DLS(tetangga, target, limit - 1)
    → found
```

```

function IDS(src: vertice, target: vertice, max_depth: integer) → boolean
{ Mengembalikan True jika target ditemukan dan
  False jika target tidak ditemukan }
KAMUS
{ Variabel }
found : boolean
i : integer
{ Fungsi dan Prosedur antara }
function DLS(src: vertice, target: vertice, limit: integer) → boolean
{ Mengembalikan True jika target ditemukan dan
  False jika target tidak ditemukan dan sudah mencapai kedalaman limit }
ALGORITMA
found ← false
i traversal [0..max_depth]
  found ← found or DLS(tetangga, target, limit)
→ found

```

Berikut adalah contoh ilustrasi kasus *folder crawling*. Misalkan terdapat struktur folder sebagai berikut.



Gambar 3.1. Contoh Struktur Folder
(sumber: Dokumen Pribadi)

Misalkan kita akan mencari file Target.txt. Berikut adalah langkah-langkah yang akan dilalui dalam penelusuran menggunakan *iterative deepening search*.

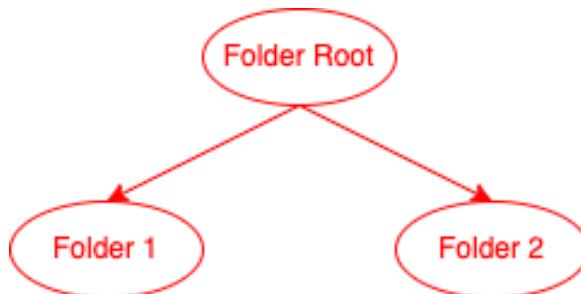
Pertama, Algoritma akan melakukan pemanggilan IDS (Folder Root, Target.txt, 2). Perhatikan bahwa max_depth bernilai 2 karena kedalaman maksimal folder adalah 3 (dimulai dari 0). Selanjutnya, algoritma akan memanggil DLS secara iteratif sebagai berikut:

1. DLS (Folder Root, Target.txt, 0)



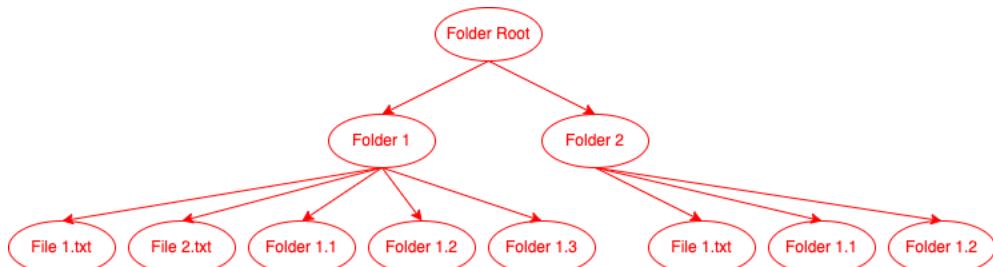
Gambar 3.2. *Iterative Deepening Search* iterasi ke-1
(sumber: Dokumen Pribadi)

2. DLS (Folder Root, Target.txt, 1)



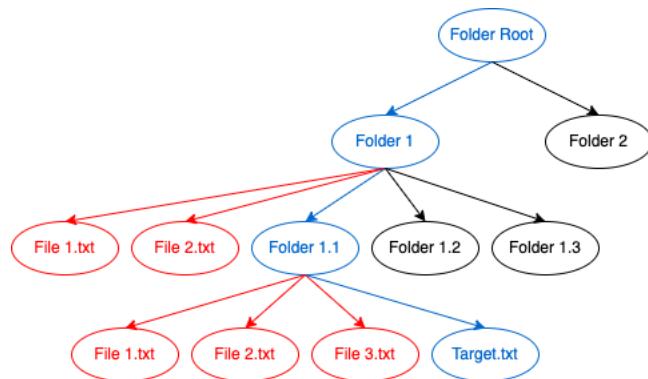
Gambar 3.3. *Iterative Deepening Search* iterasi ke-2
(sumber: Dokumen Pribadi)

3. DLS (Folder Root, Target.txt, 2)



Gambar 3.4. *Iterative Deepening Search* iterasi ke-3
(sumber: Dokumen Pribadi)

4. DLS (Folder Root, Target.txt, 3)



Gambar 3.4. *Iterative Deepening Search* iterasi ke-3
(sumber: Dokumen Pribadi)

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1. Implementasi Program

Source Code dapat diakses pada: https://github.com/davidkarelhp/Tubes2_13520065

Penjelasan program dapat diakses pada: <https://youtu.be/yXU5yzOTzYQ>

4.1.1. TraverseTreeBFS

```
procedure TraverseTreeBFS(output graph: Graph, input viewer: GViewer, input TableLayoutPanelPanel: newPanel, input root: string, input target: string, input allOccurrence: boolean)
{ Membangun pohon traversal sesuai dengan aturan BFS
  I.S. graph, root, target, allOccurrence terdefinisi
  F.S. Terbentuk pohon pencarian traversal BFS }

KAMUS
{ Variabel }
dirs : Queue
rootNode, curNode, fileNode, subDirNode : Node
curDir : string
files, subDirs : array of string
isStop : boolean
{ Fungsi dan Prosedur antara }
procedure Queue(input/output q: Queue)
{ Inisialisasi queue
  I.S. q belum terdefinisi
  F.S. q terdefinisi dan kosong }
procedure Enqueue(input/output q: Queue, input n: Node)
{ Memasukkan n ke dalam q dengan aturan FIFO
  I.S. q terdefinisi
  F.S. n masuk ke dalam q dengan aturan FIFO }
function Dequeue(q: Queue) → Node
{ Menghapus simpul dari q dengan aturan FIFO
  dan mengembalikan simpul yang dihapus}
function isQueueEmpty(q: Queue) → boolean
{ Mengembalikan True jika q kosong dan sebaliknya }
procedure Node(input/output n: Node, input val: string)
{ Inisialisasi Node dengan value val
  I.S. n belum terdefinisi dan sembarang
  F.S. n terdefinisi dengan nilai val }
procedure AddNode(input/output g: Graph, input n: Node)
{ Memasukkan n ke dalam g
  I.S. g terdefinisi
  F.S. n masuk ke dalam q }
procedure getFiles(output files: array of string, input dir: string )
{ Mendapatkan semua file pada direktori dir
  I.S. dir terdefinisi
  F.S. files terisi dengan file pada direktori dir}
procedure getDirectories(output dirs: array of string, input dir: string )
{ Mendapatkan semua file pada direktori dir
  I.S. dir terdefinisi
  F.S. dirs terisi dengan directories pada direktori dir}
function isTarget (n: Node, target: string) → boolean
{ Mengembalikan True jika nama file Node sama dengan target }
```

ALGORITMA

```
{ Inisialisasi isStop }
isStop ← False
{ Inisialisasi Queue }
Queue(dirs)
{ Inisialisasi simpul }
Node(rootNode, root)

{ Mengunjungi simpul pertama (akar) }
AddNode(graph, rootNode)
Enqueue(dirs, rootNode)

{ Mengunjungi semua simpul }
while not isQueueEmpty(dirs) do
    Node(curNode, Dequeue(dirs))
    curDir ← curNode.Id
    getFiles(files, curDir)

    for setiap file di dalam files do
        Node(fileNode, file)
        { Gambar Node }
        AddNode(graph, fileNode)
        { Gambar Edge }
        if isTarget(fileNode, target) then
            { Warna Node dengan warna biru }
            if not allOccurrence then
                isStop ← True
                while not isQueueEmpty(dirs) do
                    temp ← Dequeue(dirs) { Kosongin dirs }
            else
                { Warna Node dengan warna merah }

        if not isStop then
            getDirectories(subDirs, curDir)
            if subDirs.length > 0 then
                for setiap subDir di dalam subDirs do
                    Node(subDirNode, subDir)
                    { Gambar Node }
                    AddNode(graph, subDirNode)
                    Enqueue(dirs, subDirNode)
                    { Gambar Edge }
            else
                { Warna Node dengan merah }
        { q kosong }
```

4.1.2. TraverseTreeDFS

```
procedure TraverseTreeDFS(output graph: Graph, input viewer: GViewer, input
TableLayoutPanel: newPanel, input root: string, input target: string, input allOccurence:
boolean)
{ // Desc
  I.S.
  F.S. }

KAMUS
{ Variabel }
dirs : Stack
rootNode, curNode, fileNode, subDirNode : Node
curDir : string
files, subDirs : array of string
isStop : boolean
{ Fungsi dan Prosedur antara }
procedure Stack(input/output s: Stack)
{ Inisialisasi stack
  I.S. s belum terdefinisi
  F.S. s terdefinisi dan kosong }
procedure Push(input/output s: Stack, input n: Node)
{ Memasukkan n ke dalam s dengan aturan LIFO
  I.S. s terdefinisi
  F.S. n masuk ke dalam s dengan aturan LIFO }
function Pop(s: Stack) → Node
{ Menghapus simpul dari s dengan aturan LIFO
  dan mengembalikan simpul yang dihapus}
function isStackEmpty(s: Stack) → boolean
{ Mengembalikan True jika s kosong dan sebaliknya }
procedure Node(input/output n: Node, input val: string)
{ Inisialisasi Node dengan value val
  I.S. n belum terdefinisi dan sembarang
  F.S. n terdefinisi dengan nilai val }
procedure AddNode(input/output g: Graph, input n: Node)
{ Memasukkan n ke dalam g
  I.S. g terdefinisi
  F.S. n masuk ke dalam g }
procedure getFiles(output files: array of string, input dir: string )
{ Mendapatkan semua file pada direktori dir
  I.S. dir terdefinisi
  F.S. files terisi dengan file pada direktori dir}
procedure getDirectories(output dirs: array of string, input dir: string )
{ Mendapatkan semua file pada direktori dir
  I.S. dir terdefinisi
  F.S. dirs terisi dengan directories pada direktori dir}
function isTarget (n: Node, target: string) → boolean
{ Mengembalikan True jika nama file Node sama dengan target }
```

```

ALGORITMA
{ Inisialisasi isStop }
isStop ← False
{ Inisialisasi Stack }
Stack(dirs)
{ Inisialisasi simpul }
Node(rootNode, root)

{ Mengunjungi simpul pertama (akar) }
AddNode(graph, rootNode)
Push(dirs, rootNode)

{ Mengunjungi semua simpul }
while not isStackEmpty(dirs) do
    Node(curNode, Pop(dirs))
    curDir ← curNode.Id
    getFiles(files, curDir)

    for setiap file di dalam files do
        Node(fileNode, file)
        { Gambar Node }
        AddNode(graph, fileNode)
        { Gambar Edge }
        if isTarget(fileNode, target) then
            { Warna Node dengan warna biru }
            if not allOccurence then
                isStop ← True
                while not isQueueEmpty(dirs) do
                    temp ← Pop(dirs) { Kosongin dirs }
            else
                { Warna Node dengan warna merah }

        if not isStop then
            getDirectories(subDirs, curDir)
            if subDirs.length > 0 then
                for setiap subDir di dalam subDirs do
                    Node(subDirNode, subDir)
                    { Gambar Node }
                    AddNode(graph, subDirNode)
                    Push(dirs, subDirNode)
                    { Gambar Edge }
            else
                { Warna Node dengan merah }

    { S kosong }
}

```

4.2. Struktur Data dan Spesifikasi Program

Berikut merupakan struktur data beserta spesifikasi program yang digunakan dalam membuat program:

1. Graph

Implementasi struktur data Graph menggunakan pustaka bawaan MSAGL. Penggunaan struktur data ini berfungsi untuk merepresentasikan pohon direktori. Struktur data dalam Graph merupakan agregasi dari struktur data lainnya, yaitu:

a. Node

Node memiliki beberapa atribut lainnya, yaitu Id, Label, Color, inEdges, dan outEdges. Berikut merupakan spesifikasi terperinci dari struktur data Node:

- i. Id bertipe string, berfungsi untuk mengidentifikasi Node yang berbeda.
- ii. Label bertipe string, berfungsi sebagai teks yang akan ditampilkan ketika penggambaran.
- iii. Color bertipe enum, berfungsi sebagai pemberi warna pada Node ketika penggambaran.

- iv. inEdges bertipe list of Edge yang berisi sisi-sisi yang masuk ke dalam Node.
- v. outEdges bertipe list of Edge yang berisi sisi-sisi yang keluar dari Node.

b. Edge

Edge memiliki beberapa atribut lainnya, yaitu SourceNode, TargetNode, Connection, dan Color.

Berikut merupakan spesifikasi terperinci dari struktur data Edge:

- i. SourceNode bertipe Node, berfungsi untuk merepresentasikan Node sebagai sumber dari Edge.
- ii. TargetNode bertipe Node, berfungsi untuk merepresentasikan Node sebagai tujuan dari Edge.
- iii. Connection bertipe enum, berfungsi sebagai parameter yang menandakan suatu Edge merupakan gabungan dari Graph yang telah dibentuk sebelumnya.
- iv. Color bertipe enum, berfungsi sebagai pemberi warna pada Node ketika penggambaran.

2. Queue

Implementasi struktur data Queue menggunakan *collection* pada kelas System.Collections.Generic. Queue digunakan untuk menyimpan antrian Node (nama folder/direktori) yang akan dikunjungi oleh program saat melakukan traversal dengan metode BFS (*Breadth First Search*). Berikut merupakan implementasi dari *method* yang digunakan.

- a. Enqueue, berfungsi untuk memasukkan Node ke dalam antrian dengan aturan FIFO atau *first in first out*.
- b. Dequeue, berfungsi untuk mengeluarkan Node dari antrian dengan aturan FIFO atau *first in first out*.

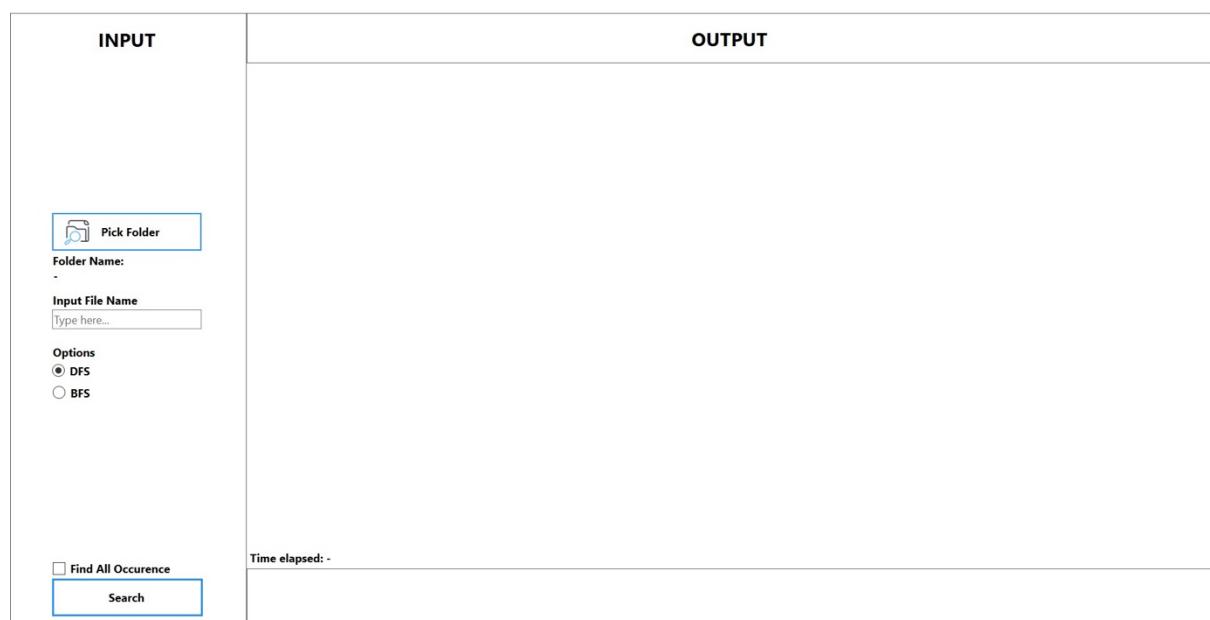
3. Stack

Implementasi struktur data Stack menggunakan *collection* pada kelas System.Collections.Generic. Stack digunakan untuk menyimpan tumpukan Node (nama folder/direktori) yang akan dikunjungi oleh program saat melakukan traversal dengan metode DFS (*Depth First Search*). Berikut merupakan implementasi dari *method* yang digunakan.

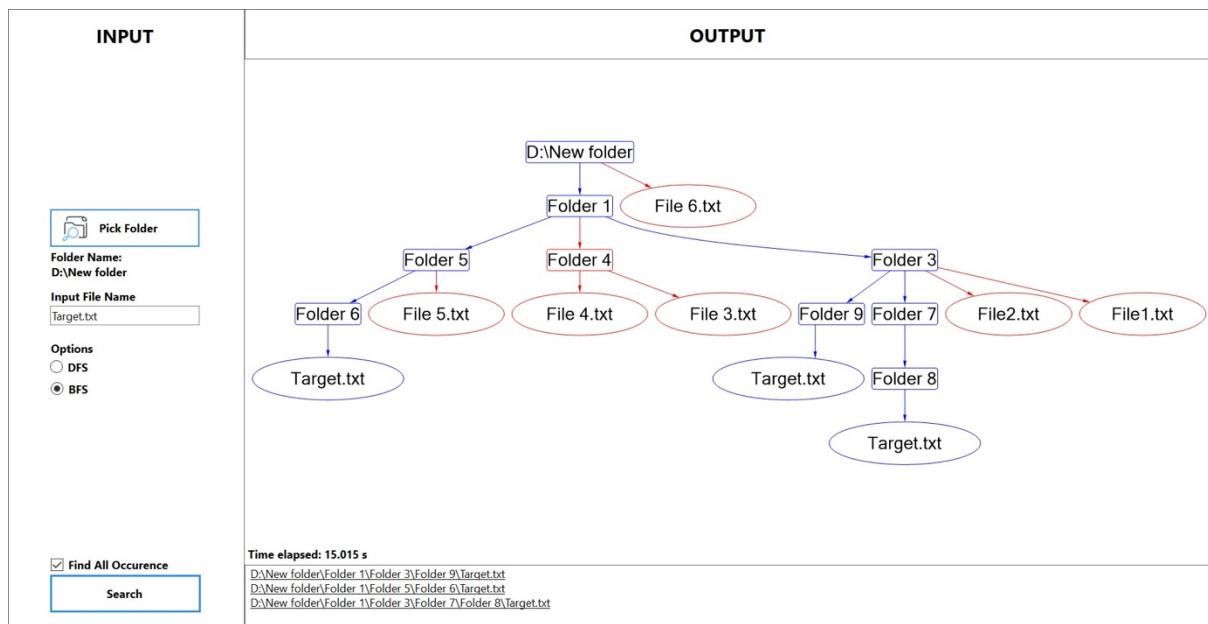
- a. Push, berfungsi untuk memasukkan Node ke dalam tumpukan dengan aturan LIFO atau *last in first out*.
- b. Pop, berfungsi untuk mengeluarkan Node dari tumpukan dengan aturan LIFO atau *last in first out*.

4.3. Tata Cara Penggunaan Program

Untuk menjalankan program, pengguna cukup menjalankan “Tubes2_13520065tttt.exe”. Setelah menjalankan program, pengguna akan diminta untuk melakukan *input* direktori awal pada **button Pick Folder**. Setelah itu, pengguna akan diminta kembali untuk melakukan *input* nama file yang ingin dicari pada bagian **Input File Name**. Selain itu, pengguna juga akan diminta untuk memasukkan algoritma yang dipilih untuk melakukan pencarian pada radio-button **Options**. Setelah melakukan *input*, pengguna dapat memilih untuk melakukan pencarian pada semua instansi dan menekan tombol **Search** untuk mulai melakukan pencarian. Pada saat pencarian, akan terdapat tampilan pembentukan *tree* sesuai dengan algoritma yang telah dipilih. Setelah pencarian berhasil, akan ditampilkan *hyperlink* untuk menuju direktori *file* yang ingin dicari. Berikut merupakan tampilan dari program atau *desktop application* yang telah kami buat.



Gambar 4.1. Tampilan Awal Program
(sumber: Dokumen Pribadi)

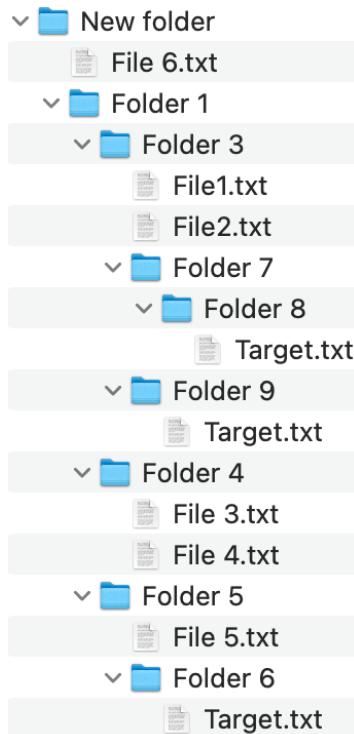


Gambar 4.2. Tampilan Akhir Program
(sumber: Dokumen Pribadi)

4.4. Hasil Pengujian

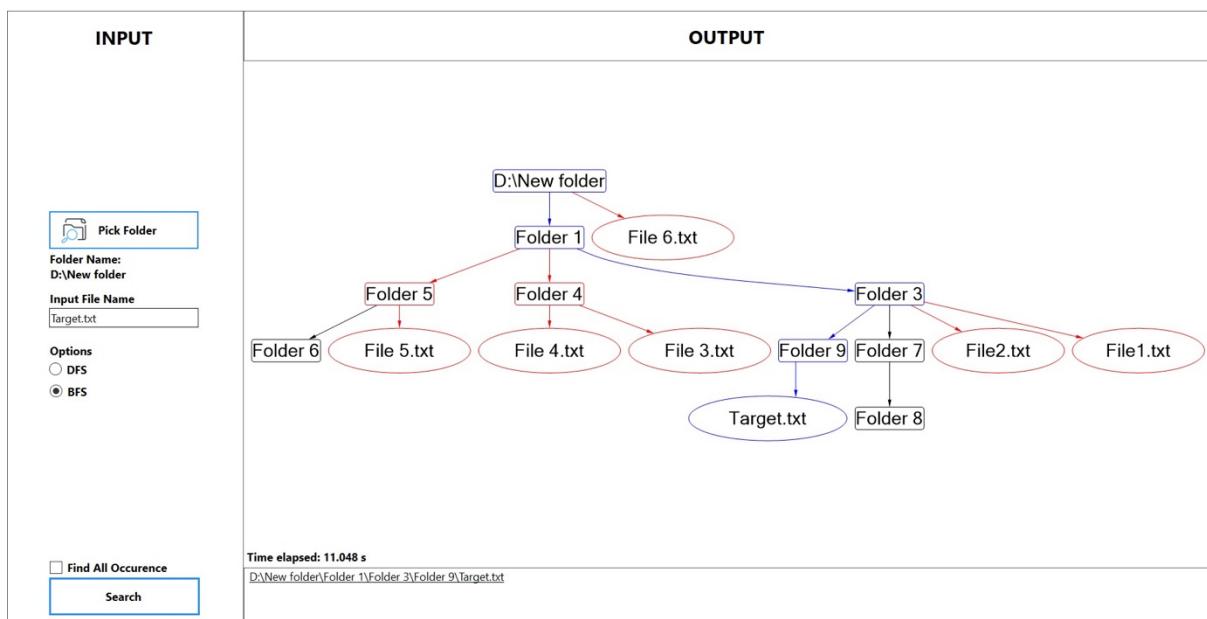
4.4.1. Pengujian 1

Berikut merupakan kasus untuk pengujian 1.



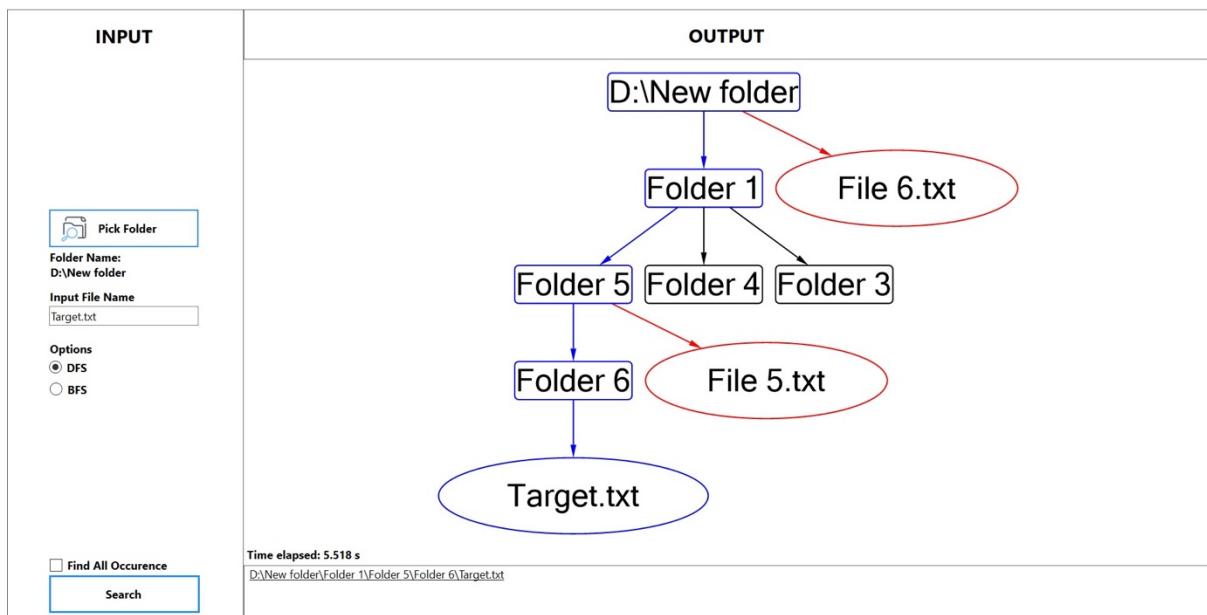
Gambar 4.3. Direktori Kasus Pengujian 1
(Sumber: Dokumen Pribadi)

Penelusuran menggunakan BFS



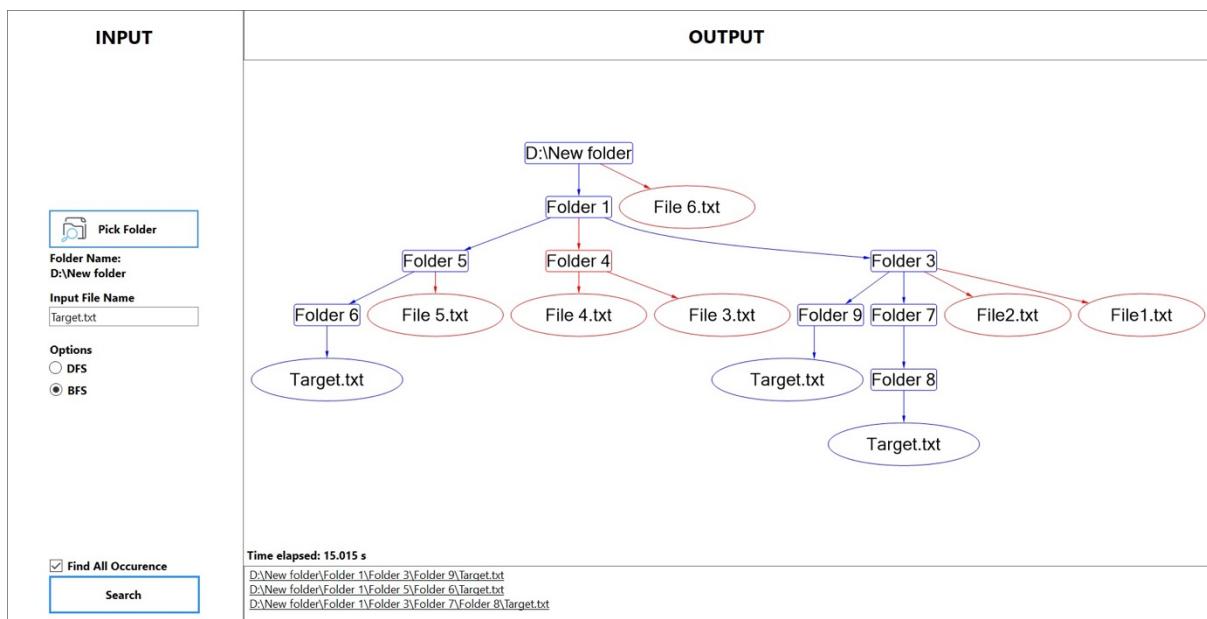
Gambar 4.4. Penelusuran Menggunakan BFS Kasus Pengujian 1
 (Sumber: Dokumen Pribadi)

Penelusuran menggunakan DFS



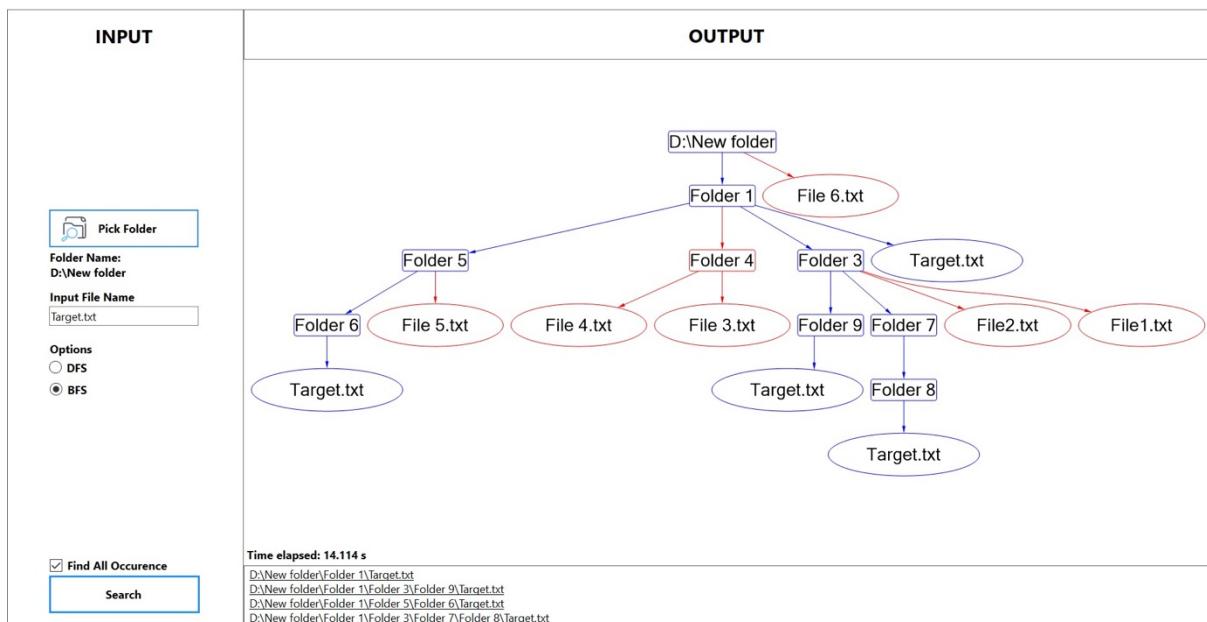
Gambar 4.5. Penelusuran Menggunakan DFS Kasus Pengujian 1
 (Sumber: Dokumen Pribadi)

Penelusuran menggunakan BFS untuk allOccurence



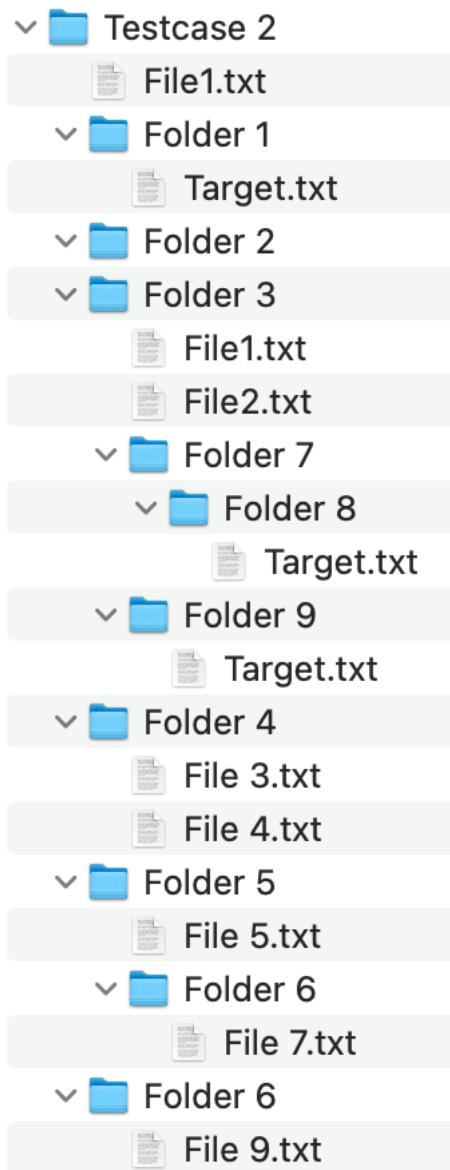
Gambar 4.6. Penelusuran Menggunakan BFS untuk allOccurence Kasus Pengujian 1
 (Sumber: Dokumen Pribadi)

Penelusuran menggunakan DFS untuk alOccurence



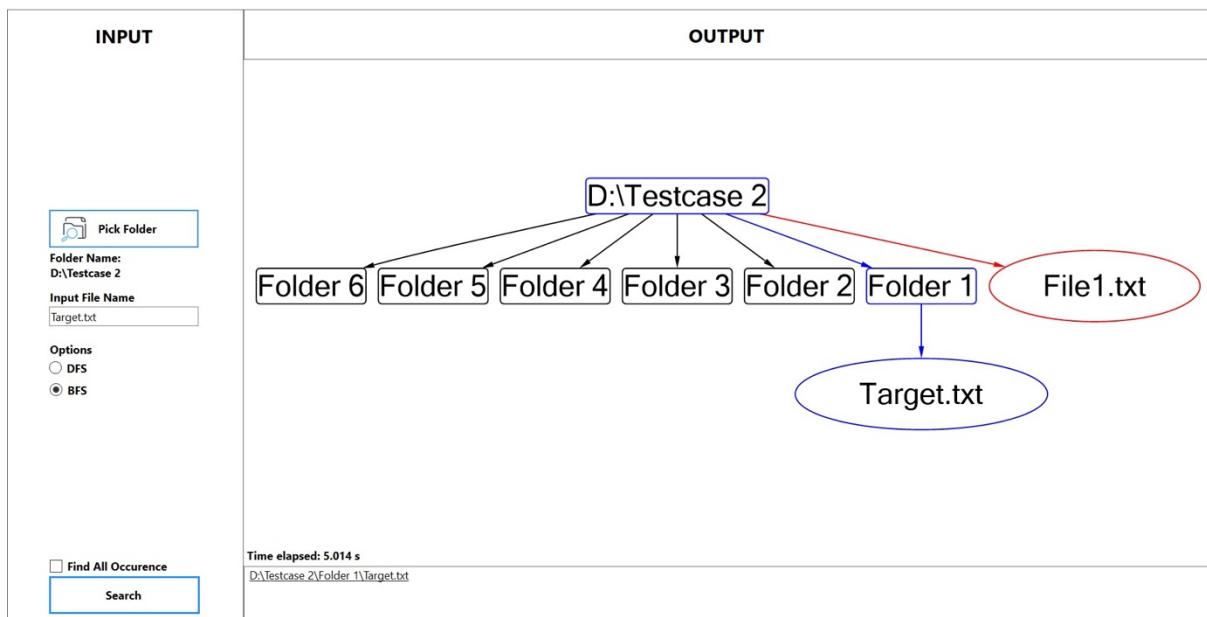
Gambar 4.7. Penelusuran Menggunakan DFS untuk allOccurence Kasus Pengujian 1
 (Sumber: Dokumen Pribadi)

4.4.2. Pengujian 2



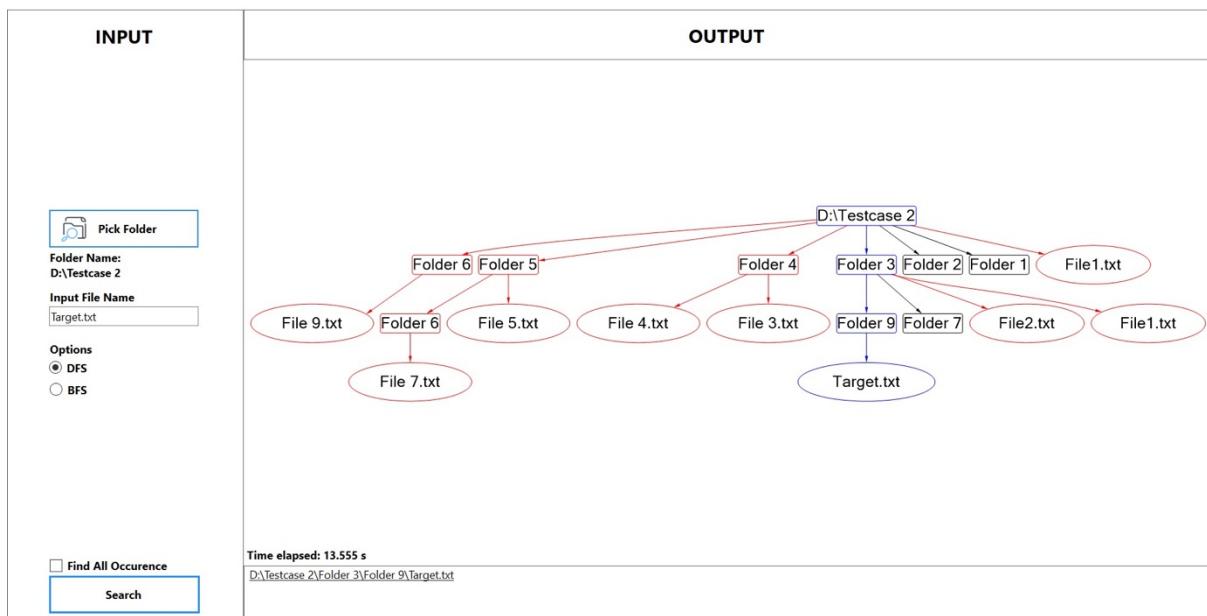
Gambar 4.8. Direktori Kasus Pengujian 2
(Sumber: Dokumen Pribadi)

Penelusuran menggunakan BFS



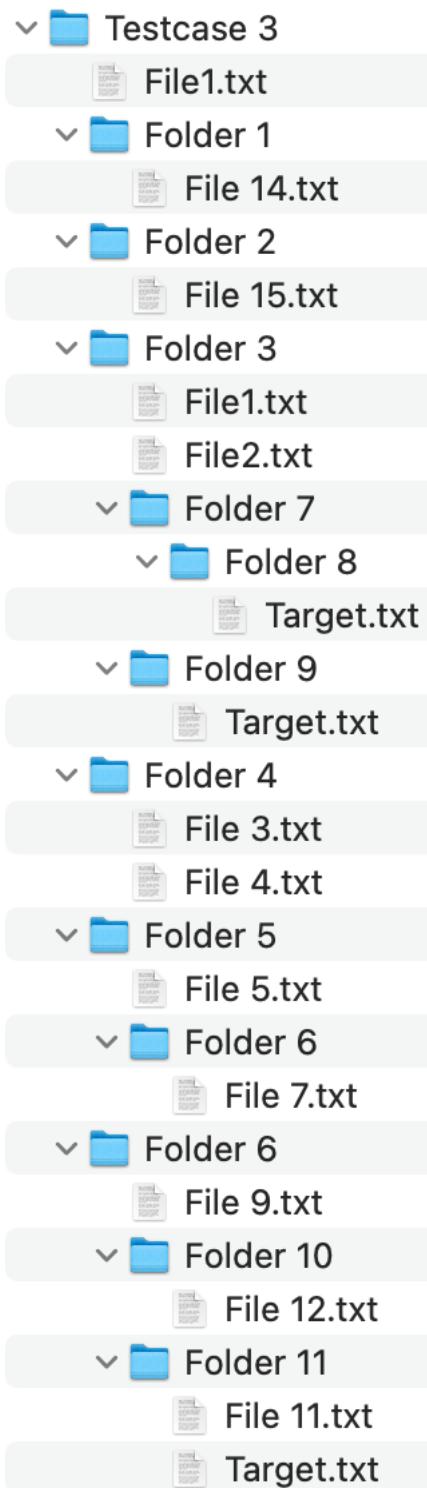
Gambar 4.9. Penelusuran Menggunakan BFS Kasus Pengujian 2
(Sumber: Dokumen Pribadi)

Penelusuran menggunakan DFS



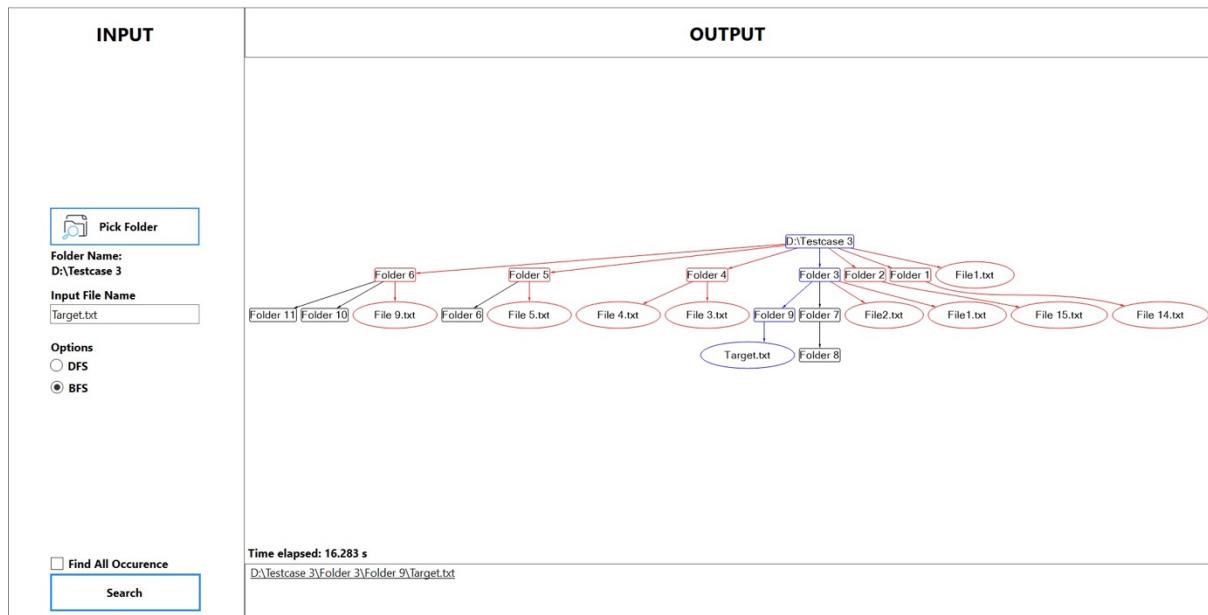
Gambar 4.10. Penelusuran Menggunakan DFS Kasus Pengujian 2
(Sumber: Dokumen Pribadi)

4.4.3. Pengujian 3



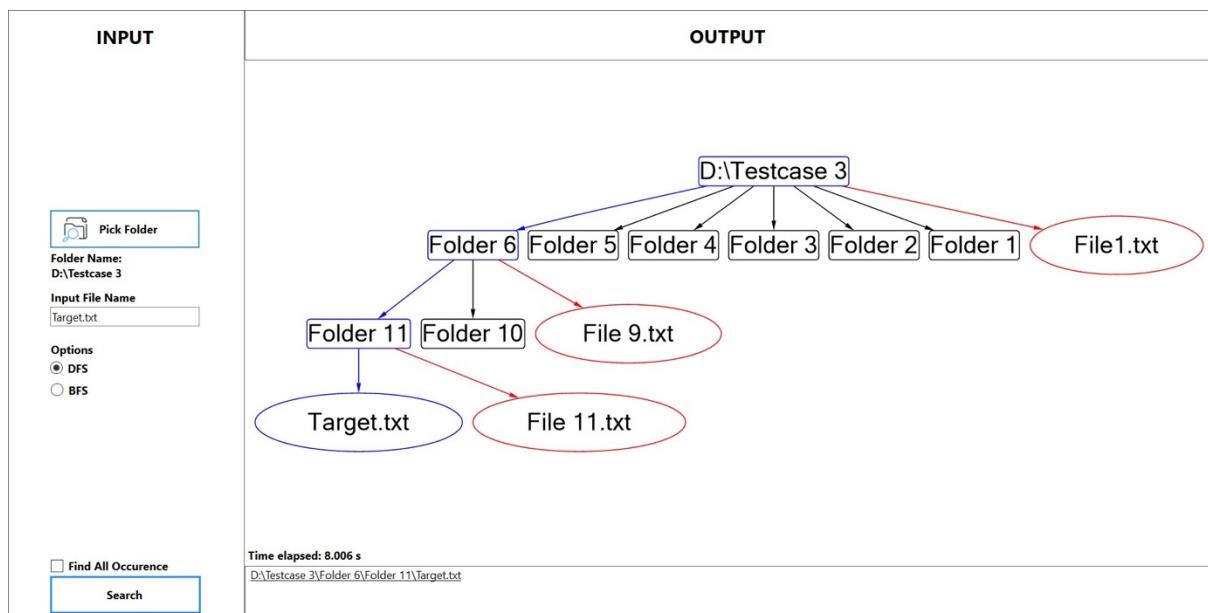
Gambar 4.11. Direktori Kasus Pengujian 3
(Sumber: Dokumen Pribadi)

Penelusuran menggunakan BFS



Gambar 4.12. Penelusuran Menggunakan BFS Kasus Pengujian 3
(Sumber: Dokumen Pribadi)

Penelusuran menggunakan DFS



Gambar 4.13. Penelusuran Menggunakan DFS Kasus Pengujian 3
(Sumber: Dokumen Pribadi)

4.5. Analisis Desain Solusi Algoritma BFS dan DFS

Berdasarkan hasil pengujian 2, algoritma BFS melalui path sebagai berikut, D:\Testcase 2 → File1.txt → Folder 1 → Folder 2 → Folder 3 → Folder 4 → Folder 5 → Folder 6 → Target.txt. Sedangkan algoritma DFS melalui path sebagai berikut, D:\Testcase 2 → File1.txt → Folder 6 → File 9.txt → Folder 5 → Folder 6 → File 5.txt → File 7.txt → Folder 4 → File 4.txt → File 3.txt → Folder 3 → Folder 9 → Target.txt. Pada kasus ini, desain solusi algoritma BFS lebih efektif dibanding solusi algoritma DFS. Jika diperhatikan, target file yang dicari dekat dengan akar direktori dari pencarian file. Sehingga, dapat disimpulkan bahwa penggunaan algoritma BFS lebih efektif digunakan pada pencarian yang dekat dengan akar.

Berdasarkan hasil pengujian 3, algoritma BFS melalui path sebagai berikut, D:\Testcase 3 → File1.txt → Folder 1 → Folder 2 → Folder 3 → Folder 4 → Folder 5 → Folder 6 → File 14.txt → File 15.txt → File1.txt → File2.txt → File 3.txt → File 4.txt → File 5.txt → File 9.txt → Folder 9 → Target.txt. Sedangkan algoritma DFS melalui path sebagai berikut, D:\Testcase 3 → File1.txt → Folder 6 → File 9.txt → Folder 11 → File 11.txt → Target.txt. Pada kasus ini, desain solusi algoritma DFS lebih efektif dibanding solusi algoritma BFS. Jika diperhatikan, pencarian menggunakan DFS menggunakan memori yang lebih sedikit dibanding BFS. Oleh karena itu, dapat disimpulkan bahwa pencarian menggunakan DFS memiliki keunggulan dalam efektifitas ruang.

BAB V

KESIMPULAN DAN SARAN

5.1. KESIMPULAN

Algoritma *breath-first search* dan *depth-first search* dapat digunakan untuk menyelesaikan permasalahan *folder crawling*. Algoritma *breath-first search* melakukan penelusuran file yang dicari secara secara menyamping atau pada level direktori yang sama terlebih dahulu, sedangkan algoritma *depth-first search* melakukan penelusuran secara mendalam atau memasuki direktori terlebih dahulu

Pengembangan *desktop application* dapat menggunakan bantuan Visual Studio dan menggunakan bahasa pemrogramman C#. Pengembangan GUI sangat terbantu dengan menggunakan WinForm.

5.2. SARAN

Pemngembangan *desktop application* menggunakan WinForm hanya dapat membuat *desktop application* yang dapat digunakan dan dikembangkan melalui sistem operasi Windows. Selain tidak *versatile*, pengembangan menggunakan WinForm menghambat penggerjaan bagi pengembang yang tidak memiliki computer dengan sistem operasi Windows. Untuk kedepannya, pengembangan *desktop application* dapat dikembangkan menggunakan pustaka yang berbeda sehingga dapat membuat *desktop application* yang lebih *versatile*.

DAFTAR PUSTAKA

- Horowitz, Ellis dkk. 1997. "Computer Algorithms". New York: Computer Science Press.
- Cormen, Thomas H. dkk. 2009. "Introduction to Algorithms". Massachusetts: MIT.
- Munir, Rinaldi, Nur Ulfa Mauladevi. 2021, "Diktat Breadth/Depth First Search", <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>, diakses pada 18 Maret 2022.
- Microsoft. "C# Documentation", <https://docs.microsoft.com/en-us/dotnet/csharp/>, diakses pada 20 Maret 2022.
- Microsoft. "Create a Windows Forms App in Visual Studio with C#", <https://docs.microsoft.com/en-us/visualstudio/ide/create-csharp-winform-visual-studio?toc=%2Fvisualstudio%2Fget-started%2Fcsharp%2Ftoc.json&bc=%2Fvisualstudio%2Fget-started%2Fcsharp%2Fbreadcrumb%2Ftoc.json&view=vs-2022>, diakses pada 18 Maret 2022.