LAPORAN TUGAS BESAR II

Pengaplikasian Algoritma BFS dan DFS dalam Implementasi Folder Crawling

Laporan dibuat untuk memenuhi salah satu tugas mata kuliah

IF2211 Strategi Algoritma

Disusun oleh:

search.me

Fayza Nadia	**13520001**
Rava Naufal A.	**13520077**
Hilda Carissa W.	**13520164**

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2022

DAFTAR ISI

DAFTAR ISI **1**

Bab 1 : Deskripsi Tugas **2**

Bab 2 : Landasan Teori **6**

2.1 Graph Traversal 6

2.2 Breadth-First Search 6

2.3 Depth-First Search 7

2.4 C# Desktop Application Development 9

Bab 3 : Analisis Pemecahan Masalah **10**

3.1 Langkah-langkah Pemecahan Masalah 10

3.1.1 Pemecahan Masalah dengan Algoritma BFS 10

3.1.2 Pemecahan Masalah dengan Algoritma DFS 10

3.2 Proses Mapping Persoalan 11

3.3 Contoh Ilustrasi Kasus Lain 12

Bab 4 : Implementasi dan Pengujian **15**

4.1 Implementasi Program Utama 15

4.2 Penjelasan Struktur Data 19

4.3 Tata Cara Penggunaan Program 23

4.4 Hasil Pengujian 25

4.5 Analisis Desain Solusi 28

Bab 5 : Kesimpulan dan Saran **30**

Kesimpulan 30

Saran 30

Link Repository Github & Video Demo **31**

Daftar Pustaka **31**

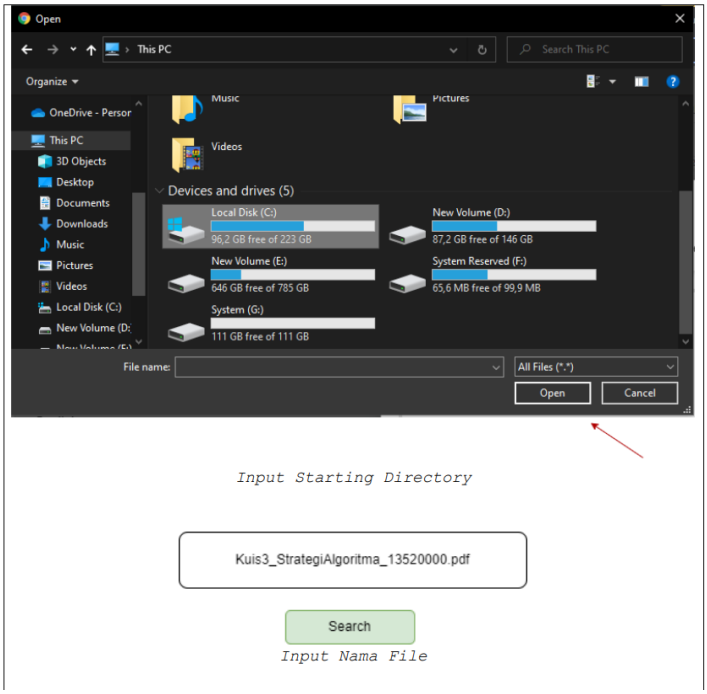
Bab 1 : Deskripsi Tugas

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.

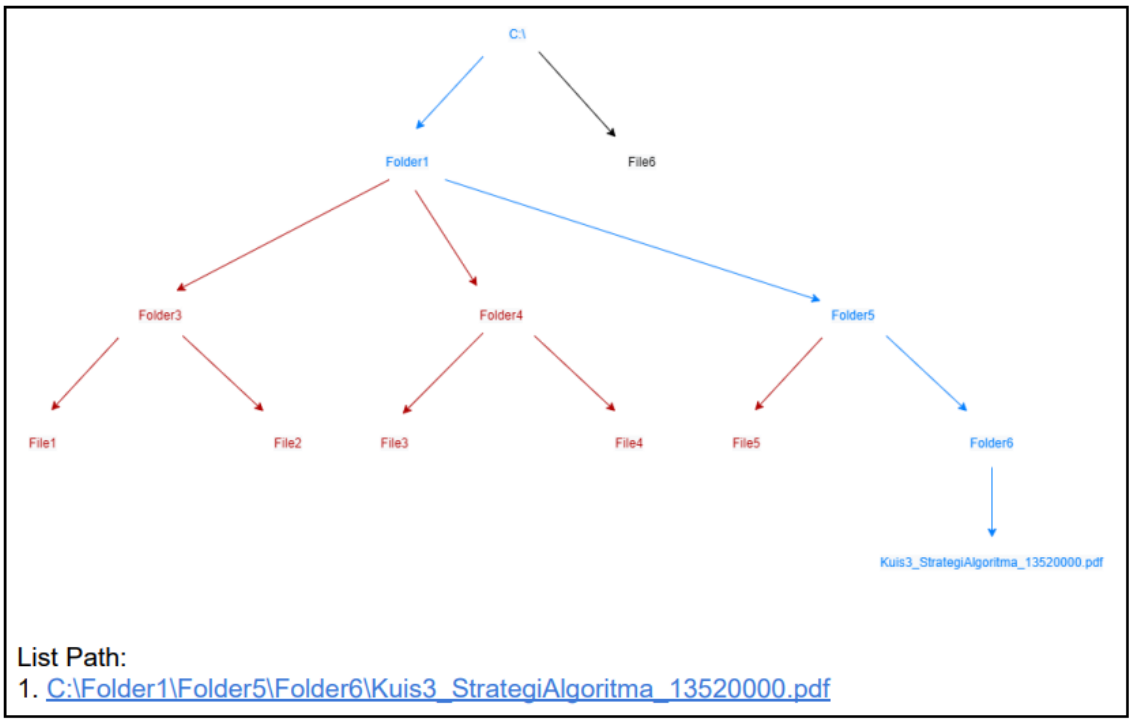
Contoh Input dan Output Program

Contoh masukan aplikasi:



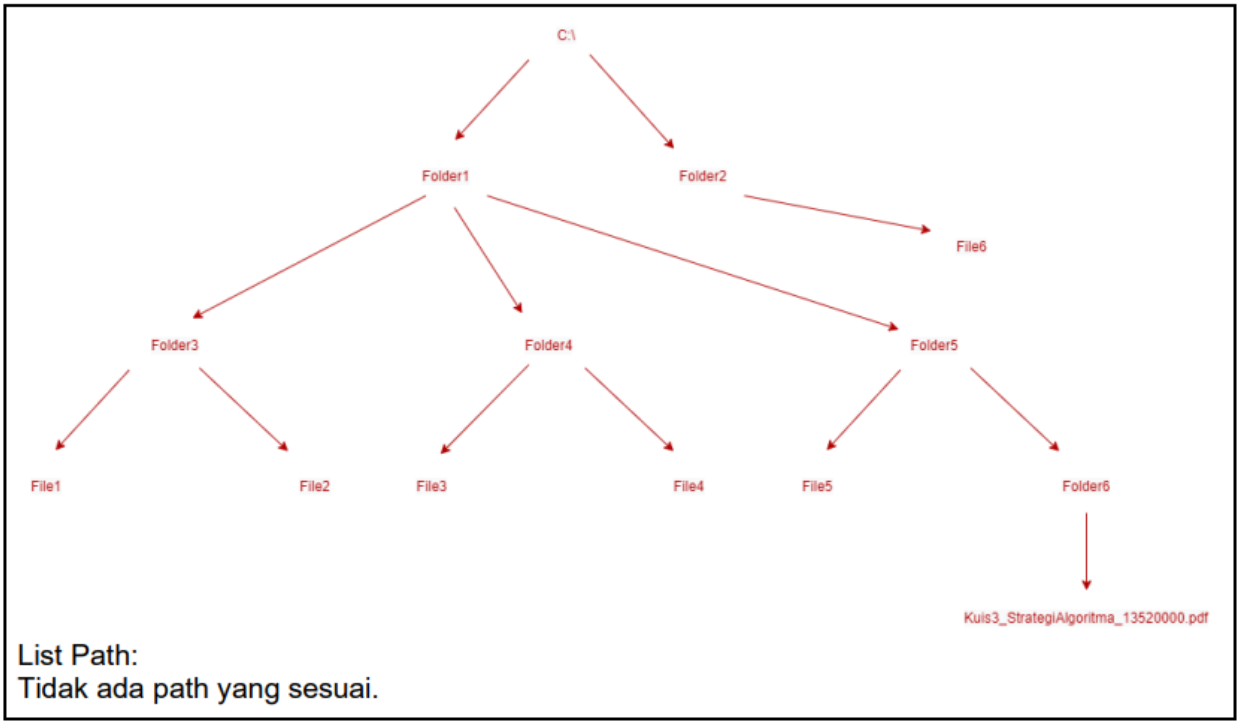
Gambar 1.1 Contoh Input

Contoh keluaran aplikasi :



Gambar 1.2 Contoh keluaran

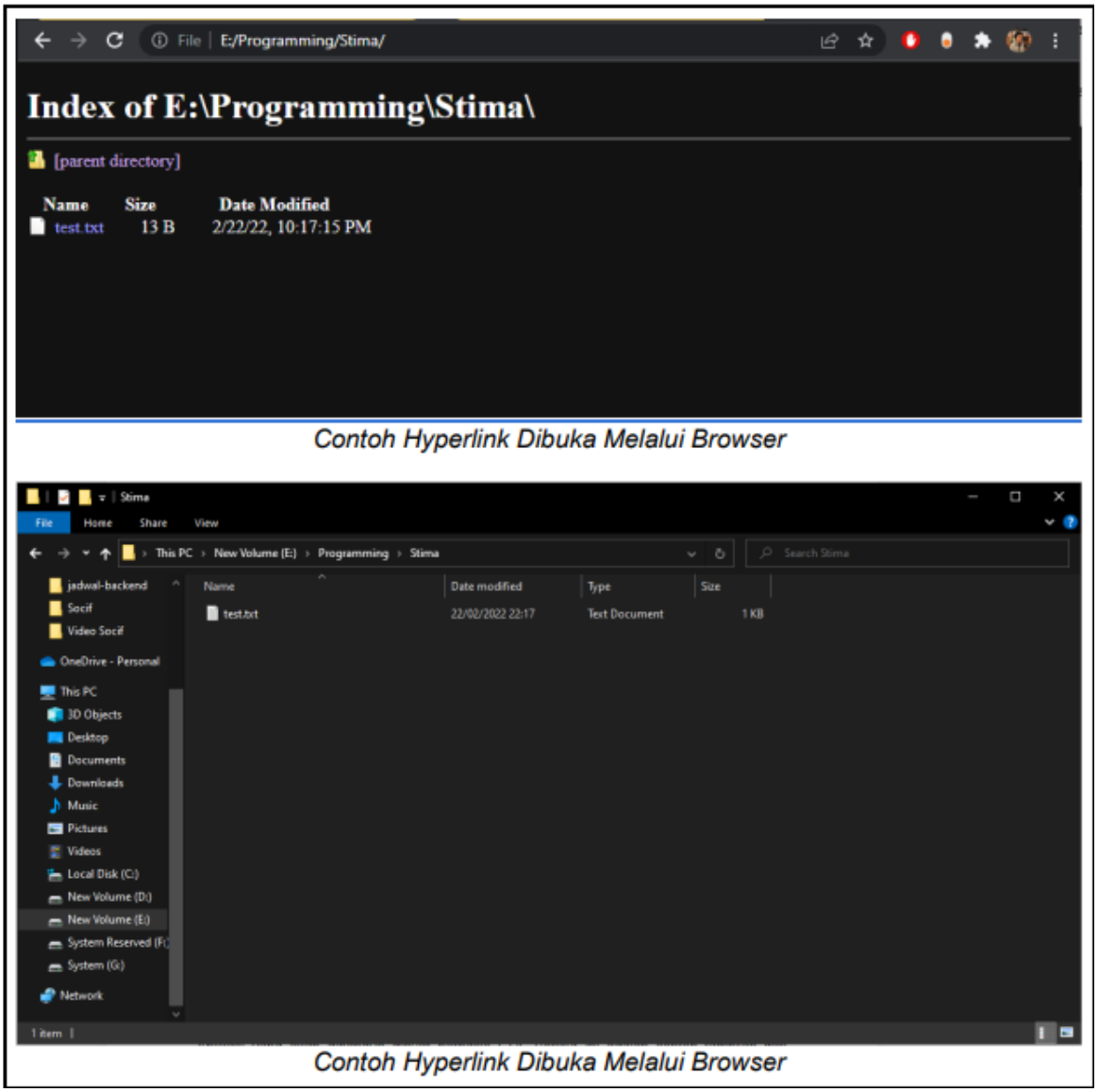
Misalnya pengguna ingin mengetahui langkah folder crawling untuk menemukan file Kuis3_StrategiAlgoritma_13520000.pdf. Maka, path pencarian DFS adalah sebagai berikut. C:\ → Folder1 → Folder3 → File1 → Folder3 → File2 → Folder3 → Folder1 → Folder4 → File3 → Folder4 → File4 → Folder4 → Folder1 → Folder5 → 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 keluaran jika 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* diklik

Bab 2 : Landasan Teori

2.1 Graph Traversal

Graph traversal adalah proses mengunjungi satu per satu simpul pada suatu graf. Proses ini dapat ditujukan baik untuk pemeriksaan maupun untuk mengubah nilai pada simpul tersebut. Pada umumnya, *graph traversal* dilakukan dengan mengunjungi simpul yang bertetangga dengan simpul awal pengecekan. *Graph traversal* sendiri dibagi menjadi dua yaitu BFS (*Breadth-First Search*) dan DFS (*Depth-First Search*) yang akan dijelaskan di sub-bab selanjutnya.

2.2 Breadth-First Search

Breadth-First Search adalah algoritma *graph traversal* yang mengunjungi setiap simpul dengan cara melebar. Algoritma Breadth-First Search dilakukan dengan mengunjungi sebuah simpul awal dan melanjutkannya dengan mengunjungi seluruh simpul yang belum dikunjungi dan bertetangga dengan simpul awal. Pada implementasinya, BFS dapat diselesaikan dengan menggunakan bantuan struktur data queue.

Berikut *pseudocode* untuk prosedur BFS pada penelusuran pohon simpul:

```
procedure BFS(input v:integer)
{ Traversal graf dengan algoritma pencarian BFS. }

  Masukan : v adalah simpul awal kunjungan
  Keluaran : semua simpul yang dikunjungi dicetak ke layar

Deklarasi
  w : integer
  q : antrian

  procedure BuatAntrian(input/output q : antrian)
  { membuat antrian kosong, kepala(q) diisi 0 }
```

```
  procedure MasukAntrian(input/output q : antrian, input v:integer)
  { memasukkan v ke dalam antrian q pada posisi belakang }

  procedure HapusAntrian(input/output q : antrian, output v:integer)
  { menghapus v dari kepala antrian q }

  function AntrianKosong(input q:antrian) → boolean
  { true jika antrian q kosong, false jika sebaliknya }

Algoritma:
  BuatAntrian(q)        { buat antrian kosong }
  write(v)              { cetak simpul awal yang dikunjungi }
  dikunjungi[v] ← true  { simpul v telah dikunjungi, tandai
                          dengan true }
  MasukAntrian(q, v)    { masukkan simpul awal kunjungan ke
                          dalam antrian }
  { kunjungi semua simpul graf selama antrian belum kosong }
  while not AntrianKosong(q) do
    HapusAntiran(q, v)  { simpul v telah dikunjungi, hapus
                          dari antrian }
    for tiap simpul w yang bertetangga dengan simpul v do
      if not dikunjungi(w) then
        write(w)        { cetak simpul yang dikunjungi }
        MasukAntrian(q, w)
        dikunjungi[w] ← true
      endif
    endfor
  endwhile { AntrianKosong(q) }
```

2.3 Depth-First Search

Depth first search adalah salah satu algoritma *graph traversal* yang berdasarkan kedalaman pohonnya. Simpul ditelusuri dari root kemudian ke salah satu simpul anaknya misal prioritas

penelusuran berdasarkan anak pertama yaitu yang paling kiri, maka penelusuran dilakukan terus melalui simpul anak pertama dari simpul anak pertama level sebelumnya hingga mencapai level terdalam.

Setelah sampai di level terdalam, penelusuran akan kembali ke 1 level sebelumnya untuk menelusuri simpul anak kedua pada pohon, lalu sama seperti sebelumnya ditelusuri hingga level terdalam, dan lalu kembali lagi ke penelusuran sebelumnya.

Pada implementasinya, DFS dapat diselesaikan menggunakan cara rekursif atau dengan struktur data stack. Dalam pengerjaan tugas besar ini, kelompok kami sendiri menggunakan cara rekursif.

Berikut *pseudocode* untuk prosedur DFS pada penelusuran pohon simpul:

```
procedure DFS(input v:integer)
{ Mengunjungi seluruh simpul graf dengan algoritma pencarian DFS }

Masukan : v adalah simpul awal kunjungan
Keluaran : semua simpul yang dikunjungi ditulis ke layar
Deklarasi
  v : integer
Algoritma
  write(v)
  dikunjungi[v] ← true
  for w ← 1 to n do
    if A[v,w] = 1 then { simpul v dan w bertetangga }
      if not dikunjungi[w] then
         DFS(w)
      endif
    endif
  endfor
```

2.4 C# Desktop Application Development

C# adalah bahasa pemrograman berorientasi objek buatan Microsoft yang dirancang untuk menulis program yang berjalan dalam framework .NET. C# sendiri biasa digunakan untuk mengembangkan aplikasi Windows dan sering juga digunakan dalam penulisan program aplikasi web hingga game. Sama seperti bahasa pemrograman berorientasi objek lainnya, C# juga menerapkan konsep-konsep objek seperti *inheritance, class, polymorphism*, *encapsulation*, dan lain-lain. Dalam penggunaan bahasa C# digunakan IDE yaitu Visual Studio 2022.

Dalam pengembangan aplikasi desktop sendiri terdapat berbagai jenis template aplikasi yang disediakan Visual Studio seperti Console Application, WPF Application, Windows Form Appliccation, dan masih banyak template project lainnya. Setiap jenis memiliki kelebihan dan kekurangannya masing-masing yang sesuai dengan kebutuhan pengguna. Untuk pengerjaan tugas besar ini, digunakan Windows Form Application untuk merealisasikan GUI (*Graphical User Interface*) dari program yang dibuat.

Bab 3 : Analisis Pemecahan Masalah

3.1 Langkah-langkah Pemecahan Masalah

Pemecahan masalah diawali dengan penetapan sebuah *folder* yang akan menjadi batasan lingkup dari pencarian *graph traversal*. Setelah itu, diberikan juga sebuah *string* dari nama *file* yang akan dicari. Berikutnya, pemilihan *searching method* antara BFS ataupun DFS akan membangun graf *folder crawling* seiring berjalannya algoritma. Dalam hal ini, *folder* digambarkan sebagai sebuah *node* atau simpul yang bersisian dengan *item* di dalamnya, baik *file* maupun sub-*folder*. Simpul-simpul tersebut kemudian akan diberikan pewarnaan yang sesuai dengan hasil pencarian antara biru, merah, atau hitam. Pada mode *find all occurence*, *graph traversal* akan tetap berjalan hingga seluruh *item* pada *root folder* telah diperiksa walaupun *file* yang dicari sudah ditemukan sebelumnya.

3.1.1 Pemecahan Masalah dengan Algoritma BFS

Penerapan algoritma BFS pada permasalahan ini diawali dengan melakukan pengecekan terhadap seluruh *file* yang terdapat di dalam *directory*. Jika *file* yang dituju tidak terdapat pada *folder* tersebut, pengecekan akan dilanjutkan pada *folder* yang terdapat di dalam *directory*. Pengecekan *subfolder* dalam hal ini adalah melakukan pengecekan pada *file* yang terdapat di dalam *folder* tersebut. Algoritma BFS kemudian dilanjutkan dengan melakukan pemeriksaan pada *folder* setelahnya pada *directory* awal. Jika seluruh *folder* pada *directory* awal sudah diperiksa, pengecekan akan dilanjutkan pada *subfolder* dari *folder-folder* tersebut tanpa memeriksa *folder* di dalam *subfolder* sebelum seluruh *file* terkunjungi. Kondisi akhir dicapai jika *file* yang dituju berhasil ditemukan atau seluruh *folder* telah dikunjungi.

3.1.2 Pemecahan Masalah dengan Algoritma DFS

Penerapan algoritma DFS pada permasalahan ini diawali dengan melakukan pengecekan terhadap seluruh *file* yang terdapat di dalam *directory*. Jika *file* yang dituju tidak terdapat pada *folder* tersebut, pengecekan akan dilanjutkan pada *folder* yang terdapat di dalam *directory*. Pengecekan *subfolder* dalam hal ini adalah melakukan pengecekan pada *file* yang terdapat di

dalam *folder* tersebut. Algoritma DFS kemudian dilanjutkan dengan melakukan pemeriksaan pada *folder* di dalam *subfolder*. Hal ini akan berlanjut hingga mencapai simpul ujung. Jika seluruh *item* baik *file* maupun *subfolder* dalam suatu *folder* telah dikunjungi, perlu dilakukan *backtracking* ke *parent folder* untuk dilakukan hal yang sama lagi . Kondisi akhir dicapai jika *file* yang dituju berhasil ditemukan atau seluruh *folder* telah dikunjungi.

3.2 Proses Mapping Persoalan

Permasalahan *folder crawling* ini dapat dipetakan sebagai berikut:

1. Operator

   Menambahkan *subpath* atau menghapus *subpath*
2. Akar

   Path yang menjadi *root folder* pencarian
3. *Problem state*

   Mencari simpul *file* yang sesuai dengan nama *file* target yang dituju
4. *Solution state*

   Path yang mengarah ke *file* target yang dituju atau tidak ditemukan sama sekali
5. *Solution space*

   Himpunan dari semua *solution state* yang ada
6. *State space*

   Semua *path* yang terdapat pada graf

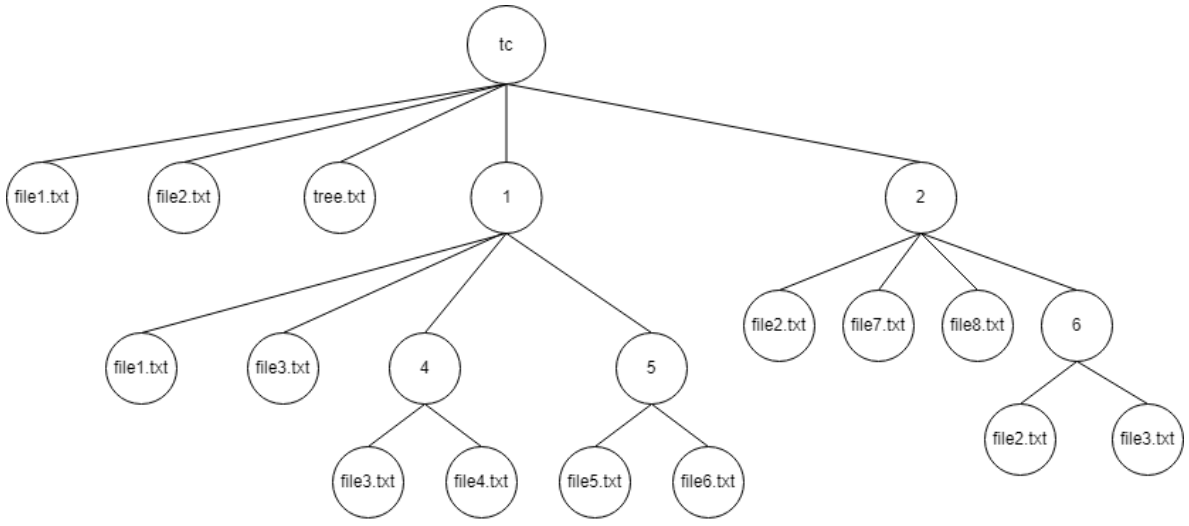
3.3 Contoh Ilustrasi Kasus Lain

Contoh directory file test case :

```
D:.
¦   file1.txt
¦   file2.txt
¦   tree.txt
¦
+---1
¦   ¦   file1.txt
¦   ¦   file3.txt
¦   ¦
¦   +---4
¦   ¦       file3.txt
¦   ¦       file4.txt
¦   ¦
¦   +---5
¦           file5.txt
¦           file6.txt
¦
+---2
    ¦   file2.txt
    ¦   file7.txt
    ¦   file8.txt
    ¦
    +---6
            file2.txt
            file3.txt
```

Gambar 3.1 Directory File Test Case

Visualisasi graf pohon dari directory file yang dipilih :

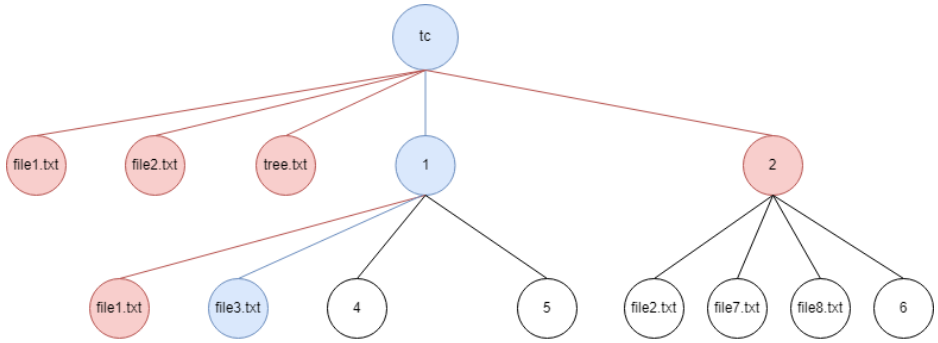


Gambar 3.2 Visualisasi Graf Directory File

Untuk pencarian file3.txt dengan BFS dan satu file terdekat saja, maka output yang diharapkan adalah sebagai berikut :

Path menuju folder parent :
`D:\..\tc`

Graf BFS



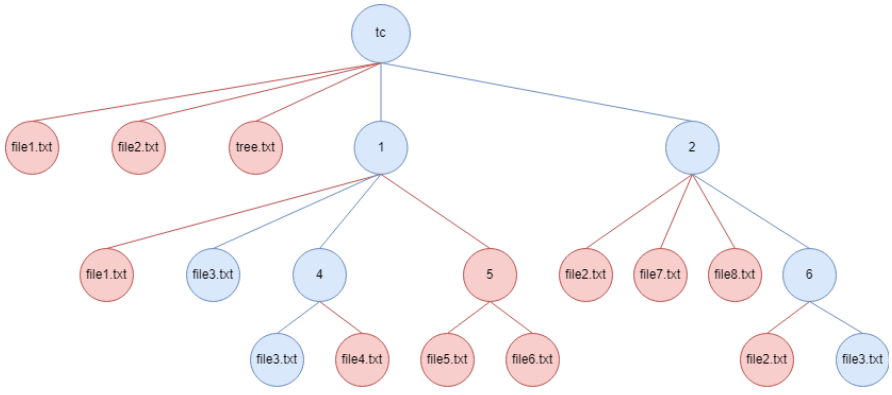
Gambar 3.3 Output Pencarian File dengan BFS

Dengan warna merah menandakan path yang sudah dicari namun tidak ditemukan, warna biru menandakan path dimana file ditemukan, dan warna hitam untuk menandakan path yang sudah masuk antrian namun belum dicari.

Sedangkan untuk pencarian file3.txt dengan BFS dan menginginkan semua kemunculan, maka output yang diharapkan adalah sebagai berikut

Path menuju folder parent :
`D:\..\tc`

Graf BFS



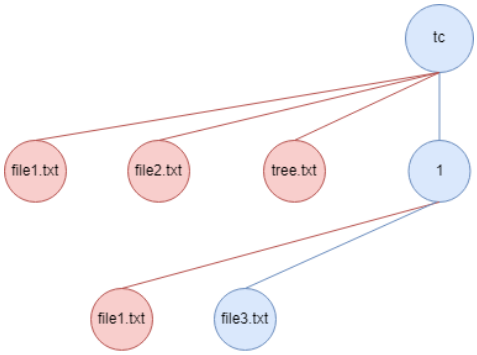
Gambar 3.4 Output pencarian file dengan BFS

Dengan warna merah menandakan path yang sudah dicari namun tidak ditemukan, warna biru menandakan path dimana file ditemukan.

Selanjutnya untuk pencarian menggunakan DFS dengan pencarian satu file saja, maka maka output yang diharapkan adalah sebagai berikut :

Path menuju folder parent :
`D:\..\tc`

Graf DFS



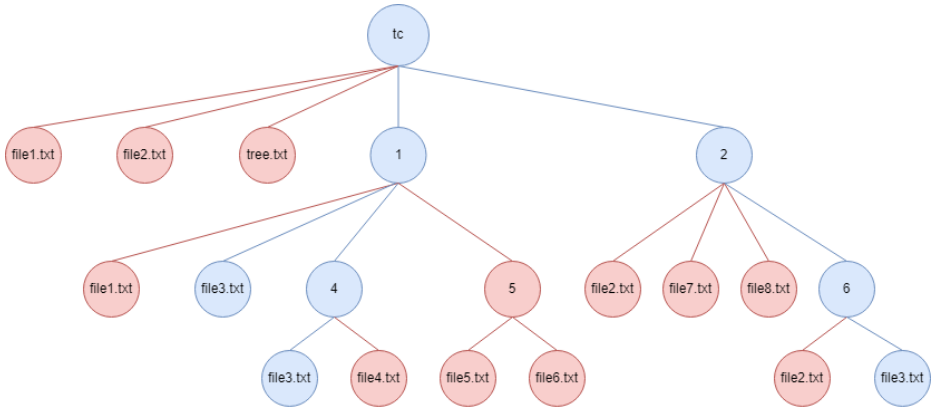
Gambar 3.5 Output pencarian file dengan DFS

Dengan warna merah menandakan path yang sudah dicari namun tidak ditemukan, warna biru menandakan path dimana file ditemukan.

Terakhir untuk pencarian menggunakan DFS dengan pencarian seluruh kemunculan file, maka maka output yang diharapkan adalah sebagai berikut :

Path menuju folder parent :
`D:\..\tc`

Graf DFS



Gambar 3.6 Output pencarian file dengan DFS

Dengan warna merah menandakan path yang sudah dicari namun tidak ditemukan, warna biru menandakan path dimana file ditemukan.

Bab 4 : Implementasi dan Pengujian

4.1 Implementasi Program Utama

Berikut implementasi beberapa fungsi penting dalam bentuk *pseudocode*:

```
Pseudocode Menampilkan Graph Saat Search Button di-Click

procedure searchBtn_Click(object sender, EventArgs e)
{ prosedur akan menampilkan graph pencarian, path dari file yang dicari, dan
waktu penyelesaian algoritma jika graph belum ditampilkan. Menghilangkan graph,
jika graph telah ditampilkan}
DEKLARASI
  rootPath : string
  countFinalPath : integer
  hyperlink : Hyperlink
  isBFS, isFound, showGraph : boolean
  graph : Graph
  sw : Stopwatch
  ALGORITMA
  if (rootPath.Length = 0) then
    write("You haven't filled the root path, fill it first!")
  else
    if (showGraph) then
      if (isBFS) then
        sw.Start()
        isFound ← BFS_search()
        if (isFound) then
          show(hyperlink) { menampilkan hyperlink }
        else
          write("File not found!")
        show(graph) { menampilkan graph }
      else
        sw.Start()
        isFound ← DFS_search()
```

```
        if (isFound) then
          show(hyperlink) { menampilkan hyperlink }
        else
          write("File not found!")
        show(graph) { menampilkan graph }
    else
      graph.Clear()
      hyperlink.Reset()
      sw.Reset()
```

```
Pseudocode Algoritma BFS

function BFS_search () → boolean
{ fungsi yang akan mengembalikan true jika file yang dicari ditemukan, false jika tidak }
DEKLARASI
  queue : Queue
  prefix, path, stringFile, stringDirectory : string
  parentDir : directory
  files : Array of file
  directories : Array of directory
ALGORITMA
  queue.enqueue(rootPath)
  visited.enqueue(rootPath)
  wereInQueue.enqueue(rootPath)
  while (queue.Count != 0) do
    path ← queue.dequeue()
    parentDir ← DirectoryInfo(path)
    visited.enqueue(path)
    if (parentDir.Extension.Length != 0) then
      if(parentDir.Name = fileTarget) then
        finalPath.enqueue(parentDir.FullName)
    else
      files ← parentDir.GetFiles()
```

```
      directories ← parentDir.GetDirectories()
      prefix ← path + '\'
      if(path[path.Length-1] = '\') then
        prefix ← path
        for (setiap file di dalam files) do
          stringFile ← Convert.ToString(file)
          if not (visited.Contains(prefix + stringFile)) then
            queue.enqueue(prefix + stringFile)
            wereInQueue.enqueue(prefix + stringFile)
        for (setiap directory di dalam directories) do
          stringDirectory ← Convert.ToString(directory)
          if not (visited.Contains(prefix + stringDirectory)) then
            queue.enqueue(prefix + stringDirectory)
            wereInQueue.enqueue(prefix + stringDirectory)
  → (finalPath.Count > 0)
```

```
pseudocode Algoritma DFS

function isFileHere(input curDir : string) → boolean
{ fungsi yang akan mengembalikan true jika file ditemukan dalam curDir, false
jika tidak }
DEKLARASI
found : boolean
file : FileInfo
fileArray : Array of file
fileName : string
ALGORITMA
  for tiap file di dalam fileArray do
    fileName ← file.ToString
    if (fileName = FileTarget) then
      finalPath.enqueue(curDir + "\\" + fileName)
      visitedFolders.enqueue(curDir + "\\" + fileName)
      ef (findAllOccurence = true) then
        Found <- true;
```

```
      else
        → true
      endif
   else
     visitedFolders.enqueue(curDir + "\\" + fileName)
   endif
  endfor
  → found

function DFS_search(input path : string) → boolean
{ fungsi yang akan mengeluarkan true jika file ditemukan, false jika tidak }
DEKLARASI
dirs : DirectoryInfo
dirArray : Array of dirs
Found : boolean
ALGORITMA
  visitedFolders ← path
  if (findAllOccurence = true) then
    isFileHere(path)
    for tiap dirs di dirArray do
      found = DFS_search(path + "\\" + dirs.ToString()
    endfor
  else
    if isFileHere(path) then
      → true
    else
      for tiap dirs di dirArray do
        found = DFS_search(path + "\\" + dirs.ToString()
        if found = true then
         → true
        endif
      endfor
    endif
  endif
  → found
```

4.2 Penjelasan Struktur Data

Berikut merupakan struktur data yang digunakan dalam program ini:

a. App.cs

Kelas App.cs berisi sekumpulan method terkait aplikasi dan responsinya dengan *user*.

Nama	Deskripsi
private string rootPath	Atribut yang berisi string *path* dari *root folder*
private string fileTarget	Atribut yang berisi string nama *file* yang ingin dicari
private bool isAllOccurence	Atribut yang mengembalikan true untuk pengecekan terhadap semua kemunculan *file*
private bool isBFS	Atribut yang mengembalikan true untuk *searching method* dengan BFS
private bool showGraph	Atribut yang bernilai true jika graf ditampilkan
public App	Fungsi konstruktor kelas App
public void setRootPath	Fungsi untuk mengatur *root path* sesuai *input*
public string getRootPath	Fungsi untuk mengembalikan string berisi *root path*
public void setFileTarget	Fungsi untuk mengatur nama *file* yang dituju
public string getFileTarget	Fungsi untuk mengembalikan string berisi nama *file* yang dituju
public void setIsAllOccurence	Fungsi untuk mengatur mode pencarian pada seluruh kemunculan *file*
public bool getIsAllOccurence	Fungsi untuk mengembalikan true jika pencarian dilakukan pada seluruh kemunculan *file*
public void setIsBFS	Fungsi untuk mengatur metode pencarian menggunakan BFS
public bool getIsBFS	Fungsi untuk mengembalikan true jika pencarian

	dilakukan dengan metode pencarian BFS
public void setShowGraph	Fungsi untuk mengatur apakah graf ditampilkan
public bool getShowGraph	Fungsi untuk mengembalikan true jika graf ditampilkan

b. BFS_Algorithm.cs

Kelas BFS_Algorithm.cs berisi atribut dan method yang digunakan dalam *graph traversal* dengan metode Breadth-First Search.

Nama	Deskripsi
private Queue<string> wereInQueue	Atribut berisi queue dari *path* yang telah masuk antrian
private Queue<string> visited	Atribut berisi queue dari *path* yang telah dikunjungi
private Queue<string> finalPath	Atribut berisi queue dari *path* akhir yang mengarah ke *file* yang dicari
private string rootPath	Atribut yang berisi string *path* dari *root folder*
private string fileTarget	Atribut yang berisi string nama *file* yang ingin dicari
private bool findAllOccurence	Atribut yang mengembalikan true untuk pengecekan terhadap semua kemunculan *file*
public BFS_Algorithm	Fungsi konstruktor kelas BFS_Algorithm
public int getVisitedCount	Fungsi untuk mengembalikan integer dari jumlah *path* yang telah dikunjungi
public int getFinalPathCount	Fungsi untuk mengembalikan integer dari jumlah *path* akhir yang mengarah ke *file* tertuju
public Queue<string> getFinalPathQueue	Fungsi untuk mengembalikan queue berisi *path* akhir yang mengarah ke *file* tertuju
public string[] getVisitedArray	Fungsi untuk mengembalikan array of string berisi *path* yang telah dikunjungi

public string[] getFinalPathArray	Fungsi untuk mengembalikan array of string berisi *path* akhir yang mengarah ke *file* tertuju
public string[] getWereInQueue	Fungsi untuk mengembalikan array of string berisi *path* yang telah masuk antrian
public bool searchOneOccurence	Fungsi untuk mengembalikan true jika pencarian dilakukan untuk satu kejadian *file* ditemukan
public bool searchAllOccurence	Fungsi untuk mengembalikan true jika pencarian dilakukan untuk seluruh kejadian *file* ditemukan
public bool BFS_search	Fungsi utama pemanggilan pencarian dengan metode BFS

c. DFS_Algorithm.cs

Kelas DFS_Algorithm.cs berisi atribut dan method yang digunakan dalam *graph traversal* dengan metode Depth-First Search.

Nama	Deskripsi
private Queue<string> visitedFolders	Atribut berisi queue dari *path* yang telah dikunjungi
private Queue<string> finalPath	Atribut berisi queue dari *path* akhir yang mengarah ke *file* yang dicari
private string FileTarget	Atribut yang berisi string nama *file* yang ingin dicari
private string rootPath	Atribut yang berisi string *path* dari *root folder*
private bool findAllOccurence	Atribut yang mengembalikan true untuk pengecekan terhadap semua kemunculan *file*
public DFS_Algorithm	Fungsi konstruktor kelas DFS_Agorithm
public int getVisitedFoldersCount	Fungsi untuk mengembalikan integer dari jumlah *path* yang telah dikunjungi
public int getFinalPathCount	Fungsi yang mengembalikan jumlah path yang berisi target file

public Queue<string> getFinalPathQueue	Fungsi untuk mengembalikan queue berisi *path* akhir yang mengarah ke *file* tertuju
public string[] getVisitedArray	Fungsi untuk mengembalikan array of string berisi *path* yang telah dikunjungi
public string[] getFinalPathArray	Fungsi untuk mengembalikan array of string berisi *path* akhir yang mengarah ke *file* tertuju
public bool isFileHere	Fungsi untuk mengembalikan true jika *file* berada pada di *directory*
public bool DFS_search	Fungsi utama pemanggilan pencarian dengan metode DFS

d. MyGraph.cs

Kelas MyGraph.cs berisi atribut serta method yang digunakan dalam pembentukan graf sesuai *output* dari algoritma BFS maupun DFS.

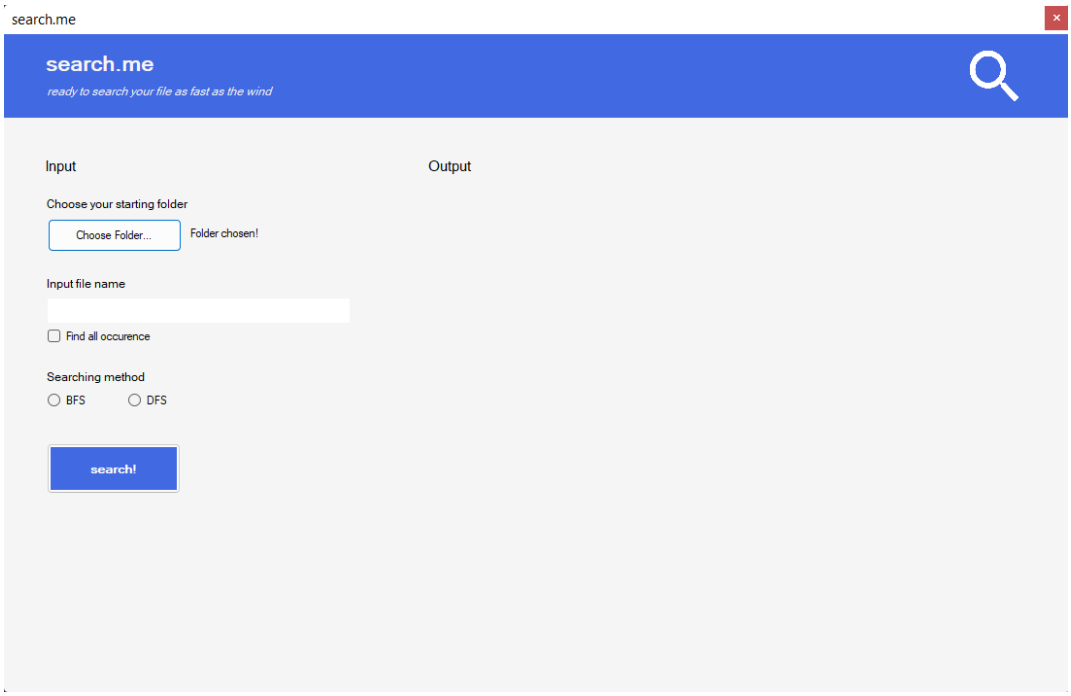
Nama	**Deskripsi**
private string rootPath	Atribut yang berisi string *path* dari *root folder*
private string[] wereInQueue	Atribut yang berisi array of string dari *path* yang telah masuk antrian
private string[] visitedPath	Atribut yang berisi array of string dari *path* yang telah dikunjungi
private string[] finalpath	Atribut yang berisi array of string dari *path* akhir dari *file* tertuju
private Microsoft.Msagl.Drawing.Graph graph	Atribut yang berisi graf MSAGL
public MyGraph	Fungsi konstruktor kelas MyGraph
public void buildGraph	Fungsi untuk membangun graf dengan *folder* atau *file* sebagai simpul
public	Fungsi untuk mengembalikan graf yang telah dibangun

Microsoft.Msagl.Drawing.Graph getGraph	
public bool isWereInQueueContain	Fungsi untuk mengembalikan nilai true jika suatu *value* terdapat pada queue wereInQueue
public bool isVisitedPathContain	Fungsi untuk mengembalikan nilai true jika suatu *value* terdapat pada queue visitedPath
public bool isFinalPathContain	Fungsi untuk mengembalikan nilai true jika suatu *value* terdapat pada array finalPath

4.3 Tata Cara Penggunaan Program

Langkah-langkah dalam menggunakan program ini adalah:

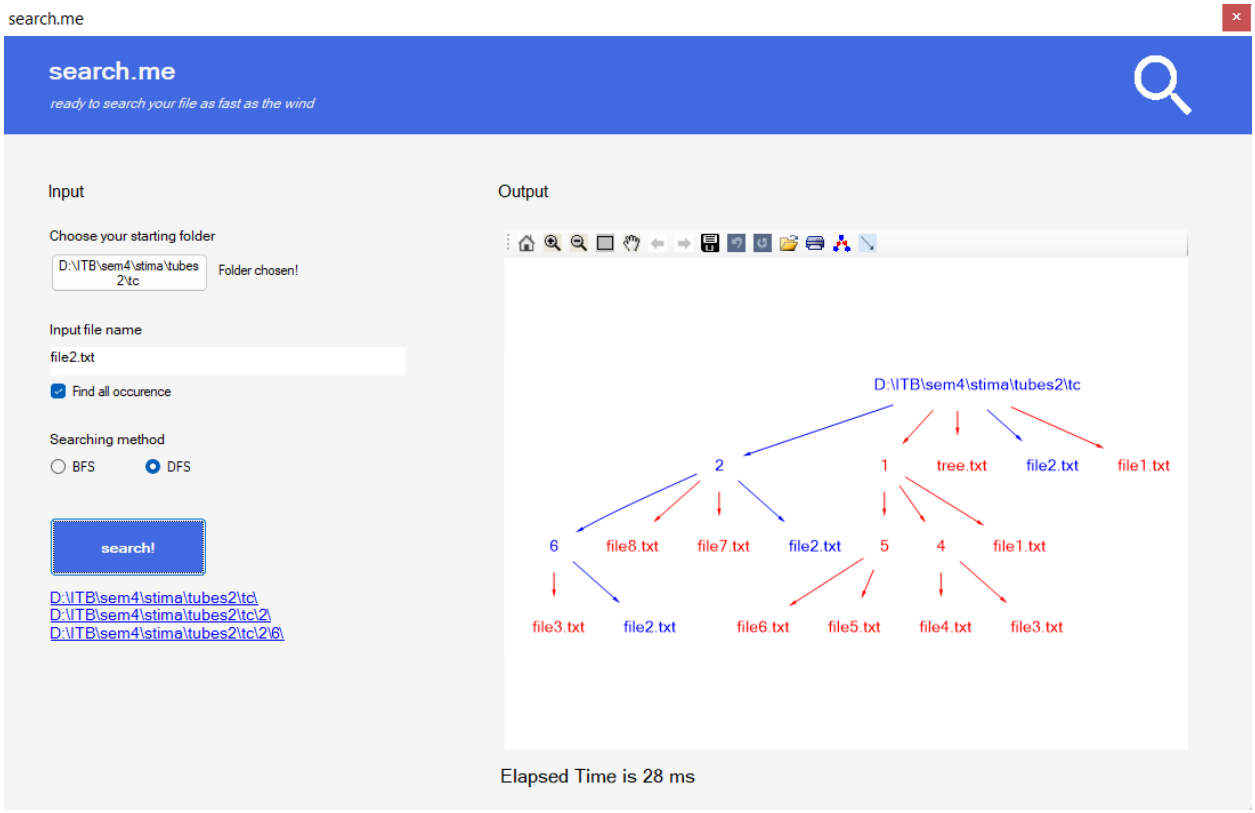
1. Membuka file "Dummy.exe" hingga muncul tampilan *interface* aplikasi berikut.



Gambar 4.1 Interface Aplikasi search.me

2. Memilih *root folder* dengan memilih tombol "Choose Folder…"

3. Memasukkan nama *file* yang ingin dicari lengkap dengan *file type*.
   Contoh: fileA.txt, docsB.docx, printC.pdf
4. Memilih satu dari dua *searching method* yang tersedia, BFS atau DFS.
5. Memilih apakah ingin mencari seluruh kemunculan. Jika ingin mencari seluruh kemunculan, maka tombol 'Find all occurence' harus diklik.
6. Menekan tombol 'search.me' untuk memulai pencarian. Hasil berupa graf yang terbentuk akan digambarkan pada panel *output* di sebelah kanan.



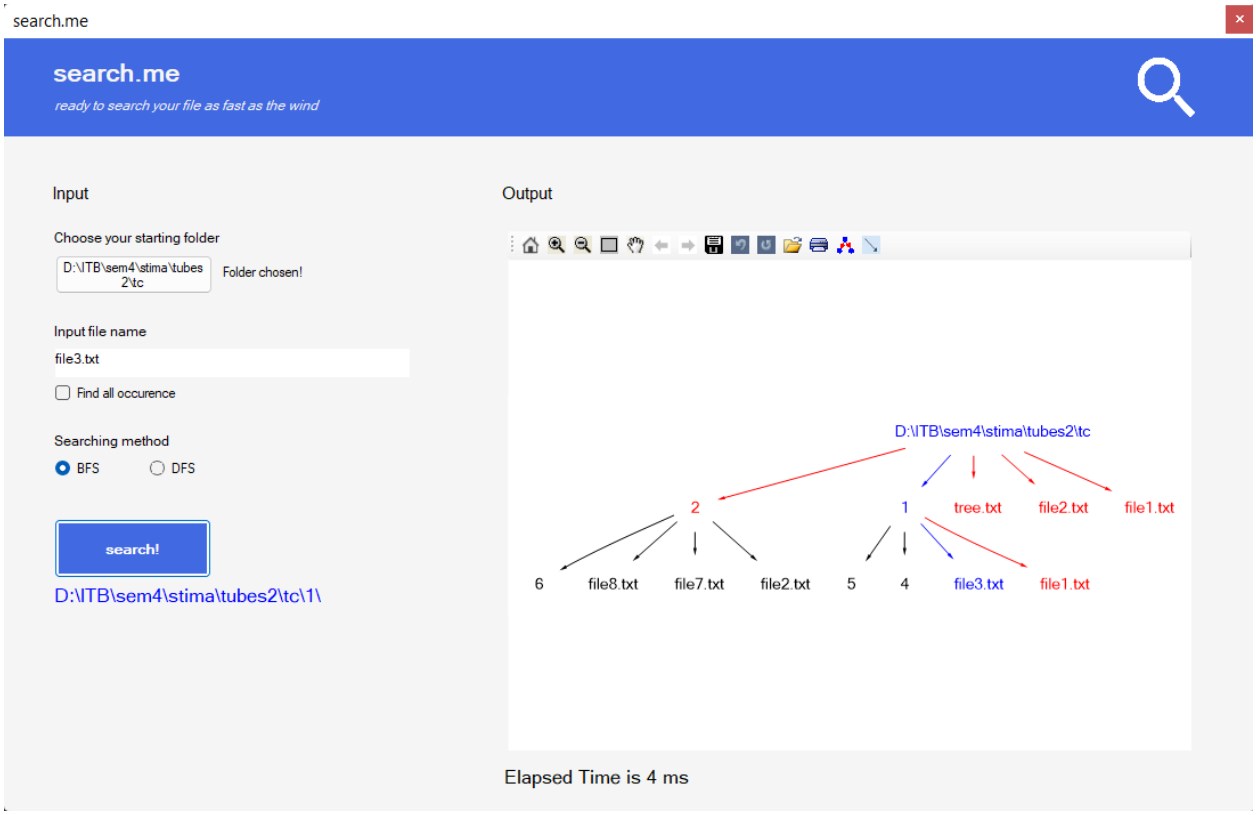
Gambar 4.2 Output Pencarian File

4.4 Hasil Pengujian

Pengujian dilakukan dengan mencari “file3.txt” pada folder utama “tc”.

a. Pengujian I

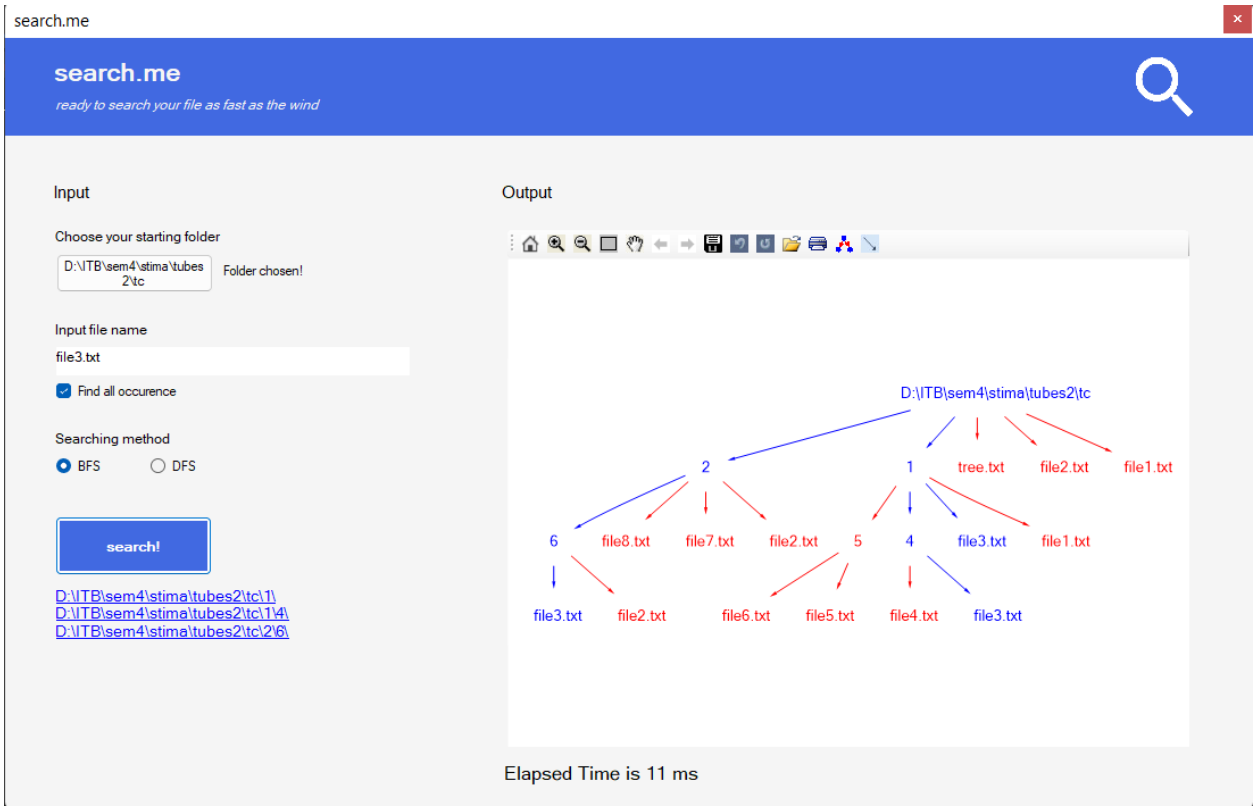
Pengujian pertama adalah pengujian Breadth-First Search tanpa mencari seluruh kemunculan.



Gambar 4.3 Hasil Pencarian 1 file dengan BFS

b. Pengujian II

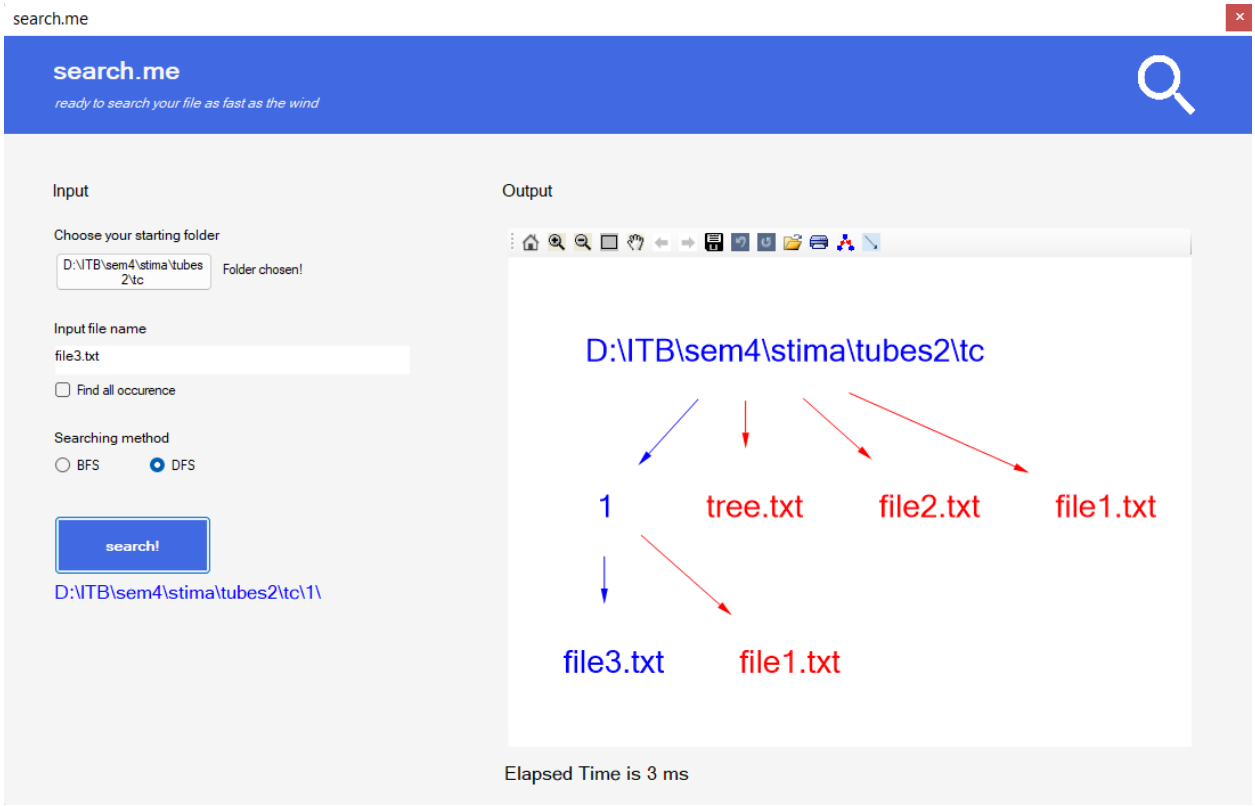
Pengujian kedua adalah pengujian Breadth-First Search dengan mencari seluruh kemunculan.



Gambar 4.4 Hasil pencarian seluruh file dengan BFS

c. Pengujian III

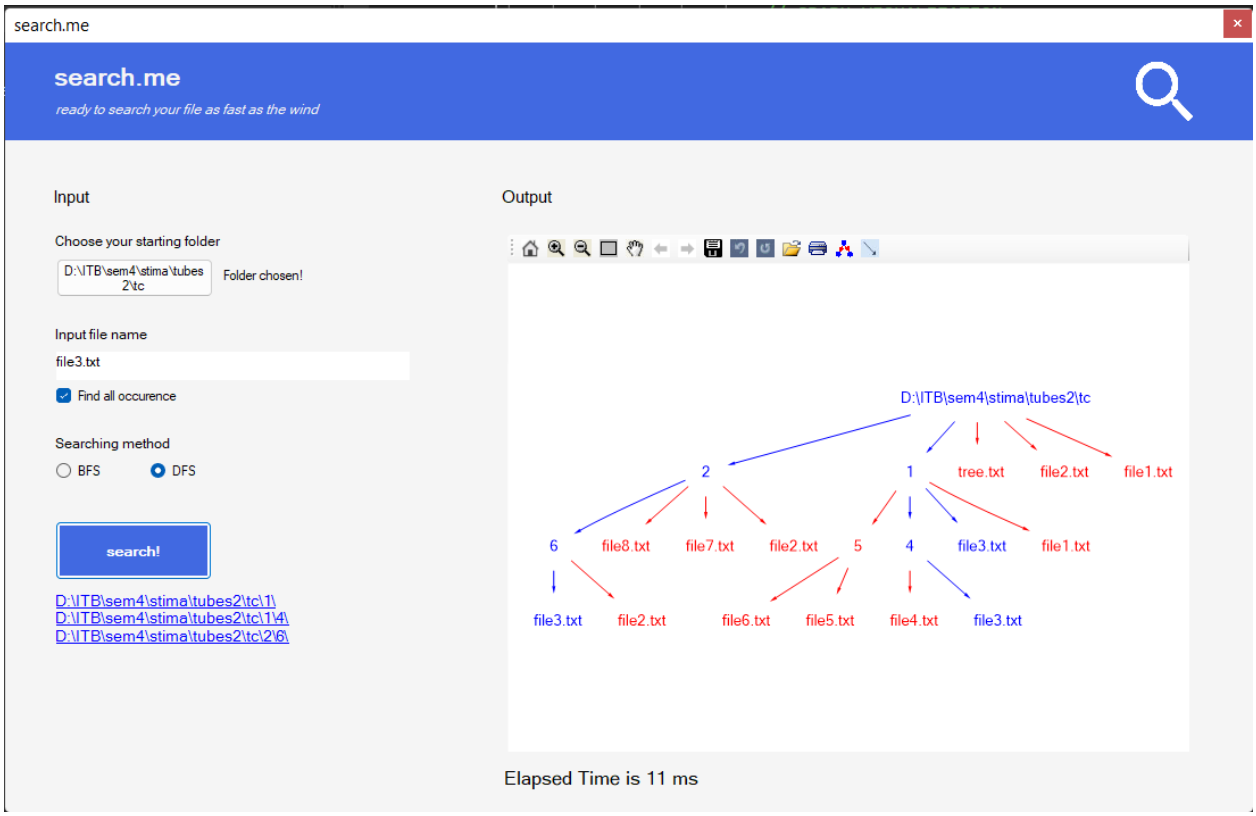
Pengujian ketiga adalah pengujian Depth-First Search tanpa mencari seluruh kemunculan.



Gambar 4.5 Hasil pencarian 1 file dengan DFS

d. Pengujian IV

Pengujian pertama adalah pengujian Depth-First Search dengan mencari seluruh kemunculan.



Gambar 4.6 Hasil pencarian seluruh file dengan DFS

4.5 Analisis Desain Solusi

Pada pengujian pertama dan ketiga, ditampilkan hasil pengujian dari pencarian *file* dengan metode Breadth-First Search dan Depth-First Search tanpa melihat seluruh kemunculan. Berdasarkan pengujian tersebut, terlihat bahwa keduanya sama-sama dapat menemukan *file* target yang ditujukan. Perbedaan antara keduanya terlihat jelas pada graf yang ditampilkan. Pada pencarian dengan Breadth-First Search, seluruh *item* di dalam *root folder* dimasukkan ke dalam antrian termasuk *folder* yang terdapat di dalamnya. Sedangkan, pada pencarian dengan

Depth-First Search, graf hanya menampilkan dua *file* saja. Karena desain dari algoritma yang mengutamakan pengecekan *file* terlebih dahulu, kedua metode pencarian tersebut, baik Breadth-First Search maupun Depth-First Search, dapat menemukan *file* target dengan waktu kurang lebih sama.

Efisiensi dari kedua metode pencarian ini sangat bergantung pada kasus uji coba. Dalam kasus di mana *file* cenderung terletak jauh di dalam *folder* yang tersembunyi di dalam *folder* lagi, algoritma Depth-First Search menjadi pilihan yang lebih baik dibandingkan dengan Breadth-First Search. Contoh kasus pada *root folder* yang memiliki banyak *subfolder* tanpa adanya *file* selain di *folder* terdalam, akan sangat tidak efisien jika menggunakan algoritma Breadth-First Search yang mencari secara meluas dan tidak mendalam. Fokus algoritma Depth-First Search yang mengutamakan pencarian hingga simpul terdalam memperbesar kemungkinan untuk menemukan *file* yang tersimpan jauh di dalam *folder* lebih cepat. Sedangkan, pada kasus di mana *file* cenderung terletak tidak jauh di dalam *folder*, algoritma Breadth-First Search menjadi pilihan metode pencarian yang tepat.

Selain dari itu, perlu diperhatikan terdapat satu faktor lain yang memegang peranan penting dalam *folder crawling* dengan kedua metode pencarian ini. Faktor ini adalah penetapan urutan memasukkan *item* ke dalam antrian pencarian. Banyak urutan yang dapat diterapkan, mulai dari prioritas sesuai abjad, ataupun tipe *file*. Dalam program yang kami buat, kedua algoritma mengutamakan untuk mengunjungi *file* terlebih dahulu dibandingkan *folder*.

Pada pengujian kedua dan keempat, ditampilkan hasil pengujian dari pencarian *file* dengan metode Breadth-First Search dan Depth-First Search tanpa melihat seluruh kemunculan. Dapat dilihat bahwa kedua metode pencarian menghasilkan graf yang sama. Hal ini dikarenakan kedua algoritma mengunjungi semua simpul *file* maupun *folder* sehingga tidak ditemukan perbedaan selain dari urutan antrian pengunjungan simpul. Sehingga, dapat dikatakan bahwa efisiensi dari kedua metode pencarian adalah sama. Hal ini seharusnya sejalan pula dengan waktu pencarian. Perbedaan selisih waktu pencarian yang sangat kecil pada kedua algoritma dapat diakibatkan karena faktor eksternal perangkat itu sendiri.

Bab 5 : Kesimpulan dan Saran

Kesimpulan

Algoritma *graph traversal*, dalam hal ini Breadth-First Search dan Depth-First Search, dapat diimplementasikan pada banyak permasalahan. *Folder crawling* merupakan salah satu dari permasalahan yang tepat untuk diimplementasikan solusinya dengan algoritma ini. Setelah dilakukan uji coba, dapat disimpulkan bahwa program yang kami buat mampu menjalankan *folder crawling* dengan menerapkan algoritma Breadth-First Search dan Depth-First Search sebagaimana yang diharapkan.

Saran

Setelah menyelesaikan tugas besar ini, saran yang dapat kami berikan antara lain adalah untuk memperdalam konsep *graph traversal* agar dapat memetakan persoalan dengan baik. Tidak hanya itu, eksplorasi terhadap *environment* IDE seperti Visual Studio dan juga *tools* MSAGL merupakan salah satu hal yang krusial pula dalam proses pengerjaan program ini. Eksplorasi yang dalam dapat pula menghemat waktu pengerjaan tugas besar ini.

Link Repository Github & Video Demo

Link Repository : hcarissa/tubes2_13520001 (github.com)

Link Video Demo : https://youtu.be/P1O5YtTeukU

Daftar Pustaka

Munir, R. (2020). Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1) (Versi baru 2021)[PDF]. Institut Teknologi Bandung. Diakses pada 22 Maret 2022 pukul 11.21 WIB melalui https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf

Munir, R. (2020). Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 2) (Versi baru 2021)[PDF]. Institut Teknologi Bandung. Diakses pada 22 Maret 2022 pukul 11.22 WIB melalui https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf