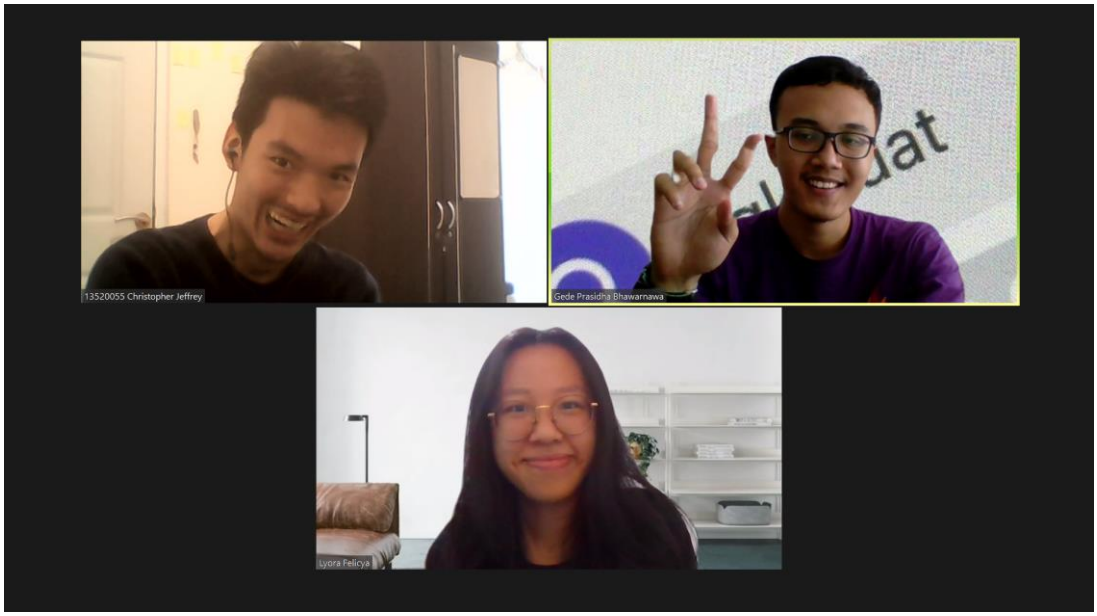
LAPORAN TUGAS BESAR

IF2211 STRATEGI ALGORITMA

PENGAPLIKASIAN ALGORITMA BFS DAN DFS DALAM IMPLEMENTASI *FOLDER CRAWLING*



Disusun oleh :

Kelompok 35 – tubes.rawr

Gede Prasidha Bhawarnawa 13520004

Christopher Jeffrey 13520055

Lyora Felicya 13520073

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2022

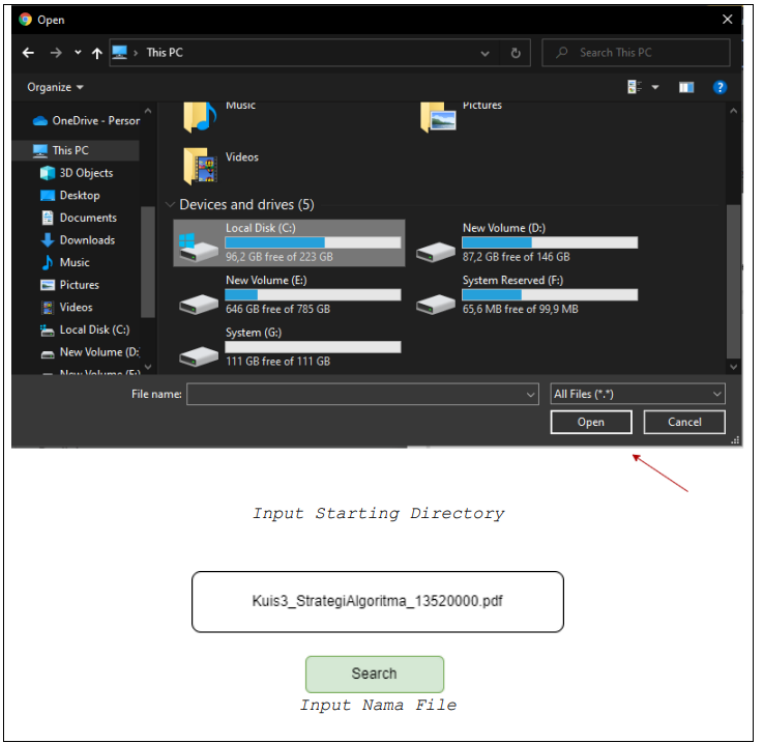
Daftar Isi

BAB 1 – DESKRIPSI TUGAS

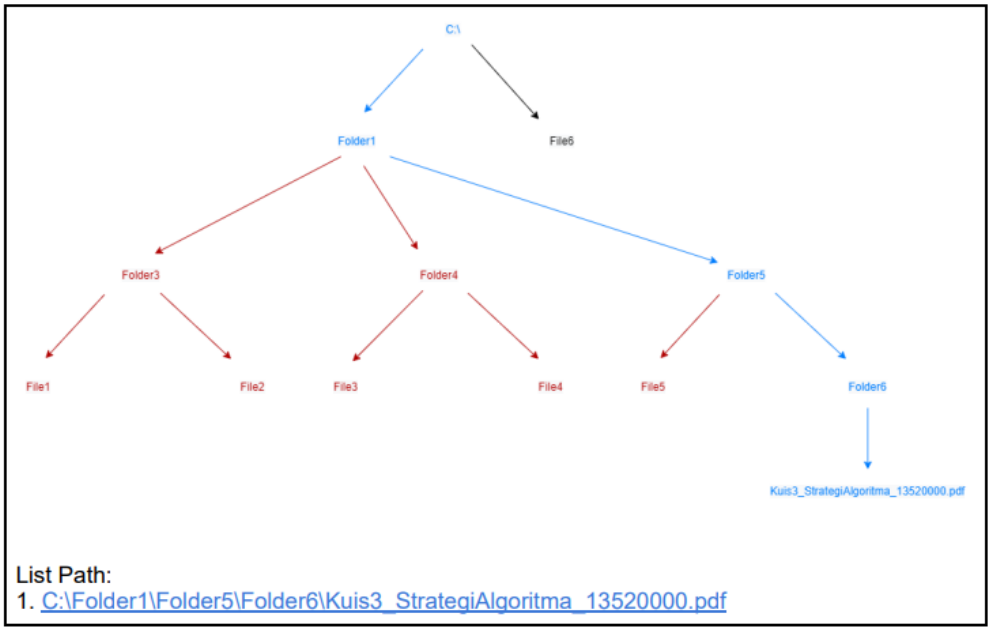
Dalam tugas besar ini, Anda akan diminta untuk membangun sebuah aplikasi GUI sederhana yang dapat memodelkan fitur dari file explorer pada sistem operasi, yang pada tugas ini disebut dengan Folder Crawling. Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS), Anda dapat 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 input aplikasi :



Contoh output aplikasi :



BAB 2 – LANDASAN TEORI

2.1 Graph Traversal

Graf merupakan sebuah bentuk representasi persoalan dimana objek-objek dibuat menjadi sebuah struktur yang saling terkait. Keterkaitan antarobjek pada sebuah graf dapat dilihat dari asal-usul objeknya, atribut (seperti penamaan, jenis/klasifikasi objek, dll), atau hubungan kepemilikan. Dalam menganalisis sebuah graf, umum digunakan metode traversal untuk mencari objek yang diinginkan. Traversal di dalam graf berarti mengunjungi simpul-simpul dengan cara yang sistematik dan terurut. Traversal graf dapat digunakan untuk mencari jalur dalam suatu graph dari titik asal ke titik tujuan, mencari jalur terpendek antara dua node/vertex, menemukan semua jalur yang bisa dilalui dari titik asal ke titik tujuan, atau untuk memecahkan persoalan optimasi pembobotan jalur pada sebuah graf. Terdapat beberapa metode yang dapat digunakan untuk melakukan traversal pada sebuah graf, seperti *breadth-first search* (BFS), *depth-first search* (DFS), algoritma A*, *depth-limited search* (DLS), *iterative deepening search* (IDS), dan algoritma dijkstra. Untuk tugas besar ini, metode traversal yang didalami dan diimplementasikan adalah algoritma BFS dan DFS.

Dalam proses traversal suatu graf, terdapat dua jenis graf yang dapat ditelusuri: graf statis dan graf dinamis. Graf statis adalah graf yang sudah terbentuk dan terdefinisi dengan jelas sebelum proses pencarian traversal dilakukan. Umumnya graf statis direpresentasikan dalam bentuk struktur data dalam sebuah program yang sudah terdefinisi. Sementara itu, graf dinamis adalah graf yang terus berubah dan diperbarahui selama proses pencarian dilakukan, dan dalam beberapa kasus masih terus berubah setelah proses pencarian mengembalikan nilai yang dicari. Contoh dari graf dinamis adalah sebuah jaringan komputer yang terhubung melalui internet. Pada tugas besar ini, bentuk graf yang digunakan adalah graf statis karena folder dana file yang diiterasi diasumsikan tidak akan berubah selama proses keberjalanan pencarian.

2.2 *Breadth-First Search* (BFS)

Algoritma *Breadth-First Search* (BFS) merupakan salah satu algoritma yang dapat digunakan dalam pencarian graf. Algoritma ini umum digunakan apabila graf yang akan dilakukan pencarian traversal merupakan graf terhubung, atau graf yang semua simpulnya terhubung dengan paling tidak satu simpul lainnya. Algoritma ini juga lebih tepat

digunakan untuk kasus-kasus *blind search*, atau kasus dimana tidak diberikan informasi tambahan yang dapat menjadi basis heuristik.

Konsep algoritma BFS adalah melakukan pencarian pada simpul awal atau simpul "target" dan menambahkan nilai simpul tersebut ke dalam sebuah tipe data antrian (*queue*). Pencarian pada simpul tunggal ini akan menjadi basis dari proses perulangan ini. Jika pencarian belum berhasil, maka simpul yang sudah dicek dapat di-*pop* dari *queue* dan ke dalam queue dapat ditambahkan semua simpul yang bertetangga dengan simpul awal tadi. Lalu, proses akan diulangi dengan elemen pertama dari *queue* (*front of queue*) yang akan diperiksa. Bila objek yang dicari ditemukan, maka proses iterasi akan berhenti (namun tergantung kasus juga). Bila sampai isi *queue* kosong masih tidak ditemukan, maka objek yang dicari tidak dapat ditemukan pada graf.

Proses kerja algoritma BFS adalah sebagai berikut:

1. Traversal dimulai dari simpul v.
2. Algoritma:
    a. Kunjungi simpul v
    b. Kunjungi semua simpul yang bertetangga dengan simpul v terlebih dahulu.
    c. Kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul - simpul yang tadi dikunjungi, demikian seterusnya

2.3 *Depth-First Search* (DFS)

Algoritma *Depth-First Search* (DFS) merupakan salah satu algoritma yang digunakan dalam pencarian graf. Sama seperti algoritma BFS, algoritma ini tepat digunakan untuk pencarian pada graf terhubung dan jika kasus pencariannya adalah *blind search*.

Konsep algoritma DFS adalah melakukan pencarian melakukan pencarian pada suatu simpul, lalu melakukan pencarian pada simpul pertama yang belum pernah dikunjungi hingga solusi ditemukan atau tidak ada simpul tetangga yang belum dikunjungi. Algoritma ini menyimpan daftar simpul tetangga yang akan dikunjungi dengan menggunakan struktur data *stack*.

Proses kerja algoritma DFS adalah sebagai berikut:

1. Traversal dimulai dari simpul v.
2. Algoritma :
   a. Kunjungi simpul v
   b. Kunjungi simpul w yang bertetangga dengan simpul v.
   c. Ulangi DFS mulai dari simpul w.
   d. Ketika mencapai simpul u sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian dirunut-balik (backtrack) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul w yang belum dikunjungi.
   e. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.

2.4 C# *Desktop Application Development*

C# *Desktop Application Development* adalah sebuah kakas yang yang berbasis desktop yang mampu beroperasi tanpa menggunakan internet. Aplikasi berbasis desktop biasanya dibuat menggunakan empat bahasa pemrograman, yaitu C++, C#, Visual Basic, dan Java. C# sendiri dapat dibantu dengan *framework* .NET yang memudahkan programmer dalam melakukan pemrograman karena bentuk visual aplikasinya yang sudah langsung terlihat sehingga proses debugging menjadi lebih mudah. *Framework* .NET sendiri adalah sebuah *framework open-source* dan *cross-platform* buatan Microsoft.

Keunggulan *desktop application* yang menggunakan *framework* .NET dan bahasa pemrograman C# pada IDE Visual Studio dibandingkan jika menggunakan bahasa, *framework*, dan IDE lainnya adalah lebih memudahkan secara sintaks karena Visual Basic jauh berbeda dibandingkan dengan bahasa C++ atau Java. Selain itu, *front-end* dari aplikasi buatan *developer* dapat diedit secara langsung menggunakan fitur *drag-and-drop* tanpa perlu melakukan *coding* seperti layaknya CSS untuk *web development*. Penggunaan Visual Studio untuk aplikasi *desktop* juga memungkinkan untuk langsung di-*export* ke dalam format *executable file* (.exe).

BAB 3 – ANALISIS PEMECAHAN MASALAH

3.1 Langkah–Langkah Pemecahan Masalah

Masalah yang diberikan adalah untuk mengimplementasikan fitur dari *file explorer*, yaitu *folder crawler*, yang dimodelkan dengan sebuah aplikasi GUI sederhana. Beberapa batasan yang diberikan adalah implementasi harus menggunakan bahasa C# dan menggunakan kakas Visual Studio dengan *framework* .NET. Langkah yang kami ambil untuk memecahkan permasalahan tersebut yaitu dengan menganalisis permasalahan terlebih dahulu, dan kemudian dipetakan menjadi sebuah program. Setelah itu dilanjutkan dengan proses mendesain GUI agar indah dilihat dan nyaman digunakan.

Folder crawling adalah sebuah proses pencarian sebuah *file* pada sebuah *directory* dengan mengiterasi setiap *file* dan *sub-directory* yang berada di dalamnya sampai file ditemukan. Proses iterasi ini dapat dilakukan menggunakan dua algoritma yang sebelumnya dibahas di dalam Bab 2, yaitu *breadth-first search* (BFS) dan *depth-first search* (DFS). Setelah diketahui cara untuk mengimplementasikan solusi dari persoalan, selanjutnya dilakukan proses eksplorasi fitur-fitur pada kakas IDE Visual Studio dan *framework* .NET. Dari langkah ini, diketahui objek-objek pada Visual Studio yang esensial dan menjadi bagian dari solusi. Pada langkah ini juga dilakukan proses eksplorasi terhadap *library* MSAGL (Microsoft *Automatic Graph Layout*), sebuah kakas yang berfungsi untuk menampilkan sebuah graf pada GUI. Lalu, dilakukan proses pembangunan aplikasi dan debugging. Setelah mencapai hasil yang diinginkan, kemudian didesain GUI aplikasi agar lebih memudahkan bagi pengguna serta nyaman dan indah dilihat oleh pengguna.

3.2 Proses Mapping Persoalan Menjadi Elemen-Elemen Algoritma BFS dan DFS

Persoalan diberikan dalam bentuk sebuah *path* atau sebuah kumpulan karakter yang menunjukkan lokasi sebuah direktori dan sebuah *filename* yang ingin dicari. Solusi yang diharapkan dihasilkan dengan menggunakan algoritma traversal graf adalah keberadaan *file* pada direktori tersebut yang memiliki nama yang sama dengan *filename* yang di-*query*-kan.

Algoritma *Breadth-First Search* (BFS) yang digunakan oleh penulis menerapkan konsep pencarian solusi pada graf statis. Pencarian dimulai dari *parent node* atau *directory*

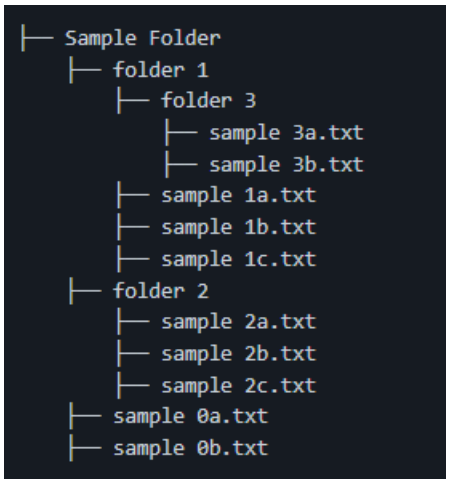
tempat dimulainya proses pencarian menggunakan *path* dari *input* pengguna, lalu diteruskan dengan mengakses semua elemen, bisa berupa direktori atau *file*. Sebelum proses pencarian dilakukan, dibentuklah sebuah tipe data *queue* atau antrian kosong pada program untuk menyimpan *path* setiap direktori dan *sub*-direktori yang ada dari titik asal. Pada awal proses pencarian, *queue* akan diisi dengan *path* direktori awal. Selanjutnya, pemeriksaan akan dilakukan dengan melakukan *pop path* pertama pada *queue* dan memeriksa *path* tersebut. Terdapat dua kemungkinan dari titik ini. Kemungkinan pertama, bila *path* yang diakses merujuk ke sebuah direktori, maka semua *path file* dan *folder* yang berada di dalam direktori tersebut harus dimasukkan ke dalam *queue*. Kemungkinan kedua, bila *path* yang diakses merupakan *path* yang merujuk ke sebuah *file*, maka akan dibandingkan dengan *filename* yang dicari. Bila sama, maka pencarian berhasil (dan dihentikan jika hanya mencari satu buah *instance*). Bila tidak, maka *path* akan dibuang dari *queue*. Untuk menandakan urutan pencarian, maka akan digunakan dua buah ID. ID yang pertama menunjukkan hubungan *parent-child* antara *folder-file* atau *folder-folder*. ID yang kedua digunakan untuk menunjukkan urutan *path file* atau *folder* yang diakses. Bila ditemukan *file* dengan nama yang sesuai dengan yang dicari, pada simpul tersebut akan diberikan akhiran "(Target)". Lalu, setelah ditemukan target *file*-nya, maka semua simpul yang mengarah ke *file* target akan diwarnai biru dan akan dihasilkan sebuah *link path* yang dapat digunakan untuk membuka *file* tersebut pada *file explorer* Windows.

Algoritma *Depth-First Search* (DFS) juga penulis implementasikan dengan menerapkan pencarian solusi pada graf statis. Sama juga seperti pada algoritma BFS, diberikan dua buah *input* dari pengguna, *path* asal direktori sebagai titik awal pencarian dan *filename* yang dicari. Pencarian juga dimulai dari *path* asal direktori titik acuan. Dari titik ini, terdapat beberapa perbedaan antara BFS dengan DFS. Bila pada BFS, *path* semua direktori dan *file* yang dapat dikunjungi disimpan di dalam sebuah *queue*, pada algoritma DFS, *path file* dan direktori disimpan di dalam sebuah tipe data *stack*. Di dalam *stack*, disimpan dua buah nilai untuk setiap entri: *path file* atau direktori, serta penanda yang dapat berbentuk bit atau boolean. Penanda ini berfungsi untuk mengetahui apakah *file* atau direktori yang dirujuk telah dikunjungi atau belum. Algoritma DFS akan terus menelusuri direktori sampai *path* yang dicari tidak dapat diderivasi lagi, atau tidak memiliki *subdirectory* lagi. Bila ujung simpul ini telah dicapai, algoritma DFS akan mengunjungi simpul tetangga (atau dalam masalah ini, *subdirectory* lain

dari *directory* awal) sampai semua *file* dan direktori telah dikunjungi. Penanganan kasus dimana *file* dengan nama yang sesuai dengan *filename* target sama dengan pada kasus algoritma BFS.

3.3 Contoh Ilustrasi Kasus Lain yang Berbeda dengan Contoh pada Spesifikasi Tugas

Sebagai metode ilustrasi, akan digunakan sebuah folder bernama "Sample Folder" dengan isinya sebagai berikut:



Berikut adalah urutan pengaksesan *file* dan *folder* beserta isi dari *queue* jika diimplementasikan algoritma BFS. Tujuan dari penggunaan tabel berisi urutan pengaksesan *file* dan *folder* adalah agar dapat dibedakan antara proses algoritma BFS dengan DFS. Asumsi yang digunakan adalah *root folder* diabaikan dan hanya *local path* saja yang digunakan pada tabel di bawah. Selain itu sebagai pembeda antara *file* dengan *folder*, digunakan akhiran (D) untuk *folder* atau *subdirectory* dan (F) untuk *file*. Isi *queue* diasumsikan sudah membaca semua *folder* dan *file* pada simpul yang diperiksa jika ada.

Urutan Pengambilan	Simpul yang Diperiksa (*Top of Queue)*	Isi *Queue* (tidak termasuk simpul yang diperiksa, hanya simpul yang belum diperiksa)
1	Sample Folder (D)	[folder 1 (D), folder 2 (D), sample 0a.txt (F), sample 0b.txt (F)]

2	folder 1 (D)	[folder 2 (D), sample 0a.txt (F), sample 0b.txt (F), folder 3 (D), sample 1a.txt (F), sample 1b.txt (F), sample 1c.txt (F)]
3	folder 2 (D)	[sample 0a.txt (F), sample 0b.txt (F), folder 3 (D), sample 1a.txt (F), sample 1b.txt (F), sample 1c.txt (F), sample 2a.txt (F), sample 2b.txt (F), sample 2c.txt (F)]
4	sample 0a.txt (F)	[sample 0b.txt (F), folder 3 (D), sample 1a.txt (F), sample 1b.txt (F), sample 1c.txt (F), sample 2a.txt (F), sample 2b.txt (F), sample 2c.txt (F)]
5	sample 0b.txt (F)	[folder 3 (D), sample 1a.txt (F), sample 1b.txt (F), sample 1c.txt (F), sample 2a.txt (F), sample 2b.txt (F), sample 2c.txt (F)]
6	folder 3 (D)	[sample 1a.txt (F), sample 1b.txt (F), sample 1c.txt (F), sample 2a.txt (F), sample 2b.txt (F), sample 2c.txt (F), sample 3a.txt (F), sample 3b.txt (F)]
7	sample 1a.txt (F)	[sample 1b.txt (F), sample 1c.txt (F), sample 2a.txt (F), sample 2b.txt (F), sample 2c.txt (F), sample 3a.txt (F), sample 3b.txt (F)]
8	sample 1b.txt (F)	[sample 1c.txt (F), sample 2a.txt (F), sample 2b.txt (F), sample 2c.txt (F), sample 3a.txt (F), sample 3b.txt (F)]
9	sample 1c.txt (F)	[sample 2a.txt (F), sample 2b.txt (F), sample 2c.txt (F), sample 3a.txt (F), sample 3b.txt (F)]
10	sample 2a.txt (F)	[sample 2b.txt (F), sample 2c.txt (F), sample 3a.txt (F), sample 3b.txt (F)]
11	sample 2b.txt (F)	[sample 2c.txt (F), sample 3a.txt (F), sample 3b.txt (F)]
12	sample 2c.txt (F)	[sample 3a.txt (F), sample 3b.txt (F)]
13	sample 3a.txt (F)	[sample 3c.txt (F)]

14	sample 3b.txt (F)	[]

Selanjutnya, berikut adalah urutan pengaksesan *file* dan *folder* beserta isi dari *stack* jika diimplementasikan algoritma DFS. Terkait asumsi dan aturan penamaan masih sama dengan ilustrasi sebelumnya, namun akan ditambahkan penomoran di belakang huruf "F" ataupun "D". Nomor yang diberikan dapat berupa "0" jika *file/folder* belum diperiksa dan "1" jika sudah diperiksa. Penambahan pada *stack* dilakukan pada *end of stack.*

Urutan Pengambilan	Simpul yang Diperiksa (*Top of Queue)*	Isi *Stack* (Untuk memudahkan pembacaan dan penulisan, hanya akan ditambahkan semua cabang simpul dari simpul yang dikunjungi. Simpul yang sudah diperiksa tidak akan ditampilkan seolah-olah sudah di-*pop*)
1	Sample Folder (D1)	[sample 0a.txt (F0), sample 0b.txt (F0), folder 2 (D0), folder 1 (D0)]
2	folder 1 (D1)	[sample 0a.txt (F0), sample 0b.txt (F0), folder 2 (D0), sample 1a.txt (F0), sample 1b.txt (F0), sample 1c.txt (F0), folder 3 (D0)]
3	folder 3 (D1)	[sample 0a.txt (F0), sample 0b.txt (F0), folder 2 (D0), sample 1a.txt (F0), sample 1b.txt (F0), sample 1c.txt (F0), sample 3a.txt (F0), sample 3b.txt (F0)]
4	sample 3b.txt (F1)	[sample 0a.txt (F0), sample 0b.txt (F0), folder 2 (D0), sample 1a.txt (F0), sample 1b.txt (F0), sample 1c.txt (F0), sample 3a.txt (F0)]
5	sample 3a.txt (F1)	[sample 0a.txt (F0), sample 0b.txt (F0), folder 2 (D0), sample 1a.txt (F0), sample 1b.txt (F0), sample 1c.txt (F0)]

6	sample 1c.txt (F1)	[sample 0a.txt (F0), sample 0b.txt (F0), folder 2 (D0), sample 1a.txt (F0), sample 1b.txt (F0)]
7	sample 1b.txt (F1)	[sample 0a.txt (F0), sample 0b.txt (F0), folder 2 (D0), sample 1a.txt (F0)]
8	sample 1a.txt (F1)	[sample 0a.txt (F0), sample 0b.txt (F0), folder 2 (D0)]
9	folder 2 (D1)	[sample 0a.txt (F0), sample 0b.txt (F0), sample 2a.txt (F0), sample 2b.txt (F0), sample 2c.txt (F0)]
10	sample 2c.txt (F1)	[sample 0a.txt (F0), sample 0b.txt (F0), sample 2a.txt (F0), sample 2b.txt (F0)]
11	sample 2b.txt (F1)	[sample 0a.txt (F0), sample 0b.txt (F0), sample 2a.txt (F0)]
12	sample 2a.txt (F1)	[sample 0a.txt (F0), sample 0b.txt (F0)]
13	sample 0b.txt (F1)	[sample 0a.txt (F0)]
14	sample 0a.txt (F1)	[]

Bila diimplementasikan fungsi pencarian pada *folder crawler*, baik itu menggunakan algoritma DFS maupun BFS, maka cukup ditambahkan fungsi pembanding *filename* bila simpul yang sedang diperiksa saat itu adalah sebuah *file*. Setelah itu, program diselesaikan dengan menuliskan waktu yang dibutuhkan untuk memproses *folder crawling* dan mengembalikan *path* yang merujuk ke *filename* yang sesuai.

BAB 4 – IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Program

Secara garis besar, berikut adalah alur dari program utama dalam bentuk pseudocode/notasi algoritma:

4.1.1 Program Utama

```
Program folderCrawler
{ Program untuk mencari sebuah file dengan filename spesifik pada sebuah folder dengan algoritma BFS dan DFS }
Kamus Data
      fileName : string
      pathDirectory : string
      { Asumsi fileName dan pathDirectory selalu valid }
      bfsOrDFS : boolean {Asumsi bila True = BFS, False = DFS}
      searchAllOccurence : boolean
      outputGraph : graph
      pathListBox : listBox
      timeRequired : real
Algoritma
      Input(fileName);
      Input(pathDirectory);
      Input(bfsOrDFS);
      Input(searchAllOccurence);
      If (bfsOrDFS) then: { masuk mode pencarian dengan BFS }
            bfsMethod(fileName, pathDirectory, searchAllOccurence);
      else:
            dfsMethod(fileName, pathDirectory, searchAllOccurence);
      Output(outputGraph);
      Output(pathListBox);
      Output(timeRequired);
```

4.1.2 Implementasi Algoritma BFS

```
procedure bfsMethod(input string fileName, input string
      pathDirectory, input boolean searchAllOccurence, output graph
      outputGraph, output listBox pathListBox, output timeRequired)
{ prosedur untuk melakukan folder crawling menggunakan algoritma BFS
      }
Kamus Data
      pathListTuple : tuple<int,int,string> {berisikan parentId,
      selfId, dan path}
      pathList : queue
      parentId : integer
      selfId : integer
      targetFile : string
      currentPath : string
      isFound : boolean
Algoritma
      parentId <- -1;
      selfId <- 0;
      enqueue(pathList, parentId, selfId, pathDirectory); {mengisi ke
      depan queue}
      while (length(pathList) ≠ 0) do:
            currentPath <- pop(pathList);
            repeat:
                  enqueue(pathList, parentId++, selfId++,
      subdirectory(currentPath));
            until (subdirectory(currentPath).IsIn(pathList) or
      subDirectory(currentPath) = NULL); {menambahkan semua
      subdirectory pada currentPath ke dalam queue}
            repeat:
                  enqueue(pathList, parentId, selfId++,
      files(currentPath));
            until (files(currentPath).IsIn(pathList) or
      files(currentPath) = NULL); {menambahkan semua file pada
      currentPath ke dalam queue}
```

```
            isFound <- checkFileIsSame(fileName,
      filename(currentPath)); { mengecek apabila file yang sedang
      diperiksa sama dengan file yang sedang dicari }
            addNode(outputGraph, parentId, selfId,
      filename(currentPath));
```

4.1.3 Implementasi Algoritma DFS

```
procedure dfsMethod(input string fileName, input string
      pathDirectory, input boolean searchAllOccurence, input integer
      parentId, input integer selfId, output graph outputGraph,
      output listBox pathListBox, output timeRequired)
{ prosedur untuk melakukan folder crawling menggunakan algoritma DFS
      }
Kamus Data
      pathListTuple : tuple<int,int,string> {berisikan parentId,
      selfId, dan path}
      pathList : stack
      parentId : integer
      selfId : integer
      targetFile : string
      currentPath : string
      isFound : boolean

Algoritma
      if (parentId = -9 and selfId = -9) then: {untuk inisiasi kasus
      pertama saat pertama kali dipanggil prosedurnya}
            parentId <- -1;
            selfId <- 0;
      push(pathList, parentId, selfId, pathDirectory); { mengisi ke
      belakang stack }
      while (length(pathList) ≠ 0) do:
            currentPath <- pop(pathList);
            repeat:
```

```
            dfsMethod(fileName, pathDirectory,
searchAllOccurence, parentId++, selfId++, outputGraph,
pathListBox, timeRequired);
            addNode(outputGraph, parentId, selfId,
filename(currentPath));
      until (subdirectory(currentPath).HasBeenVisited() or
subDirectory(currentPath) = NULL); {mengunjungi semua
subdirectory pada currentPath sampai semua telah dikunjungi
atau tidak ada subdirectory yang dimiliki oleh currentPath}
      repeat:
            addNode(outputGraph, parentId, selfId,
filename(currentPath));
      until (files(currentPath).HasBeenVisited()); {menambahkan
file currentPath ke dalam outputGraph}
      isFound <- checkFileIsSame(fileName,
filename(currentPath)); { mengecek apabila file yang sedang
diperiksa sama dengan file yang sedang dicari }
```

Fungsi-fungsi seperti enqueue, addNode, HasBeenVisited, filename, dan lainnya merupakan fungsi *built-in* yang sesuai dengan tipe data masing-masing.

4.2 Penjelasan Struktur Data yang Digunakan dalam Program dan Spesifikasi Program

Dalam program dipakai beberapa struktur data, baik itu berupa standar maupun *built-in library* yang sudah disediakan oleh bahasa pemrograman itu sendiri. Beberapa tipe data standar yang digunakan untuk kebutuhan implementasi program ini adalah *string*, *boolean, integer*, dan *float*. Tipe data standar ini digunakan untuk mengakses *path* dari setiap direktori dan *file* yang diperiksa oleh algoritma BFS maupun DFS. Selain itu, tipe-tipe tersebut juga digunakan untuk menampilkan hasil akhir kepada pengguna dalam bentuk waktu selesai dan *path* tempat lokasi ditemukannya *file*.

Namun, selain penggunaan tipe dan struktur data standar diatas, digunakan juga beberapa tipe data abstrak yang lebih kompleks. Contohnya adalah penggunaan struktur data *queue* pada

algoritma BFS dan penggunaan struktur data *stack* pada algoritma DFS. *Queue* dan *stack* pada dasarnya adalah sebuah *dynamically-allocated list*, yang artinya sebuah list berisikan objek yang dialokasikan secara dinamis. Artinya, jumlah elemen anggotanya tidak tetap dan dapat berubah-ubah selama masih bisa dialokasikan memori untuk tipe data tersebut. Selain itu, meskipun mereka berdua adalah *list*, kedua tipe data tersebut tidak dapat diakses dengan cara yang sama.

Queue adalah sebuah *list* yang menggunakan konsep FIFO (*first in, first out*), dimana proses iterasi elemen berlaku seperti antrian di dunia nyata, dimana elemen yang dapat diakses dan dapat dimodifikasi adalah elemen yang berada di indeks pertama *queue*. Selain itu, penambahan elemen ke dalam *queue* juga seperti antrian, yaitu elemen akan dimasukkan ke dalam *end of list* atau *end of queue* dan menunggu elemen berada di depan *queue* agar dapat diakses.

Sementara itu, *stack* atau tumpukan adalah sebuah tipe *list* yang mengimplementasikan konsep LIFO (*last in first out*). Artinya, elemen yang bisa diakses pada sebuah *stack* adalah elemen yang terakhir dimasukkan ke dalam *stack* tersebut. Sama juga seperti tumpukan di dunia nyata, penambahan elemen pada *stack* berada pada *end of list* atau *end of stack*.

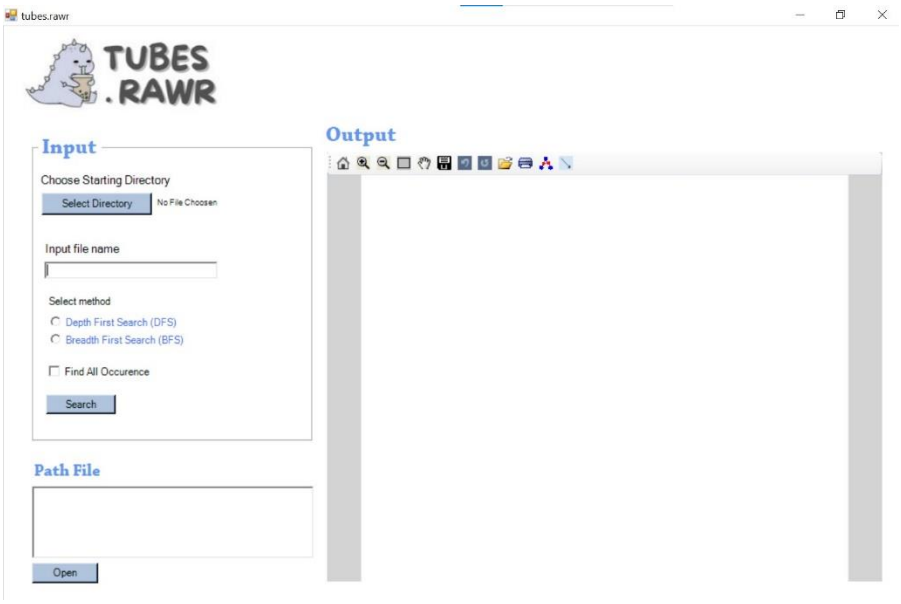
Penggunaan *queue* dan *stack* pada algoritma BFS dan DFS dilatarbelakangi proses dan langkah kerja kedua algoritma tersebut. Pada algoritma BFS, penyimpanan *path* direktori dan *file* disimpan dengan aturan FIFO karena algoritma BFS akan melakukan pemeriksaan semua elemen graf pada suatu kedalaman (*depth of graph*) secara terurut dari atas hingga bawah. Dengan menggunakan *queue*, dapat dijamin tidak akan ada elemen graf yang terlewat oleh algoritma BFS. Sementara itu, algoritma DFS melakukan pencarian secara terurut dengan "mendalami" graf, atau memulai dari simpul dengan kedalaman terkecil sampai mencapai simpul *leaf* atau daun (simpul yang tidak memiliki simpul anak atau *child node*) pada kedalaman maksimum. Setelah itu, perlahan-lahan pemeriksaan akan mengecek simpul yang lebih tinggi satu kedalaman dan tetangganya, sampai semua tetangga pada semua simpul telah dilewati. Karena itu, penggunaan *stack* lebih tepat karena pemeriksaan dilakukan secara terbalik, dalam arti dilakukan dari kedalaman terdalam hingga kedalaman terdangkal. Dalam kasus ini, lebih tepat jika traversal dilakukan dengan melakukan pengambilan *path* dari *end of list* (*end of stack*).

Selain *queue* dan *stack*, struktur data yang diimplementasikan dalam program ini adalah graf terhubung. Graf terhubung diimplementasikan menggunakan kakas MSAGL untuk menampilkan graf yang menggambarkan hubungan *parent-child* antara *folder-file*. Pada graf yang

diimplementasikan penulis, hubungan antara *parent-child* dibuktikan dengan nilai *parent-id* yang sama antara *folder* dengan semua *file* yang ada di dalam *folder* tersebut. Selain itu, digunakan juga parameter *self-id* untuk menunjukkan urutan pembangkitan simpul pada graf. Pada hubungan antara *folder* dengan *file*, hubungan antar simpul dibuat dengan melihat nilai dari *parent-id*. Untuk hubungan antara *folder* dengan *subfolder*, hubungan antar simpul dibuat dengan melihat nilai dari *parent-id* dan *self-id*, dengan *parent-id* harus berselisih satu. Alasannya adalah terdapat perbedaan kedalaman satu tingkat antara *folder* dengan *subfolder*.

Dalam menentukan spesifikasi pemakaian program ini, dapat dilihat dari proses *input/output*. Program ini memiliki beberapa objek yang dapat menerima *input* seperti *Directory Search*, *textbox*, dan *radio button*. Setelah itu, setelah melakukan pemrosesan, maka akan ditampilkan *path* semua *file* yang ditemukan di dalam direktori awal dan dimasukkan ke dalam sebuah *listbox*. Lalu, ditampilkan juga sebuah *graf* hasil penelusuran direktori dan urutan pencarian simpul pada direktori.

4.3 Penjelasan Tata Cara Penggunaan Program



Gambar 4.3.1 Tampilan Awal Program

Setelah program dijalankan, akan muncul tampilan halaman utama dari program folder crawling. Pertama, pengguna dapat memilih direktori folder yang ingin digunakan untuk pencarian. Pengguna juga harus input nama file yang ingin dicari, lengkap dengan eksistensinya, serta metode pencarian yang ingin digunakan (BFS/DFS). Pengguna juga dapat ceklis bagian Find All Occurrence untuk mencari seluruh kemunculannya. Graf pencarian akan divisualisasikan pada bagian output setelah pengguna menekan tombol Search. Path menuju file yang dicari juga akan tertulis pada tempat yang telah disediakan.

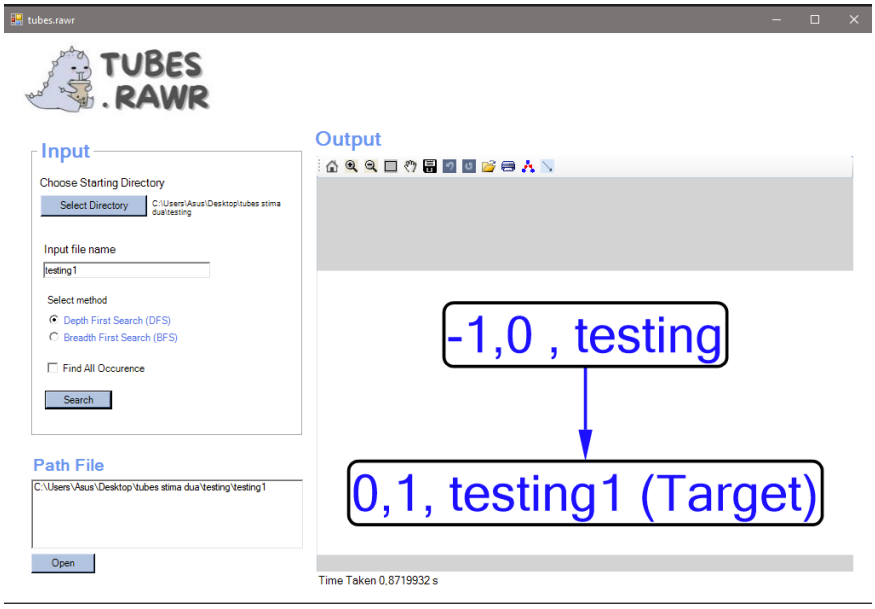
4.4 Hasil Pengujian

Berikut merupakan isi dari Folder testing yang digunakan untuk pengujian

```
testing
      testing1
            testing7
                  filekembar.txt
      testing2
            testing3
                  folderkembar
                  testing4
                        file8.txt
                        file9.txt
                  File7.txt
            testing5
                  file3.txt
                  file4.txt
                  file6.txt
                  filekembar.txt
      testing6
            folderkembar
            file2.txt
      File1.txt
```

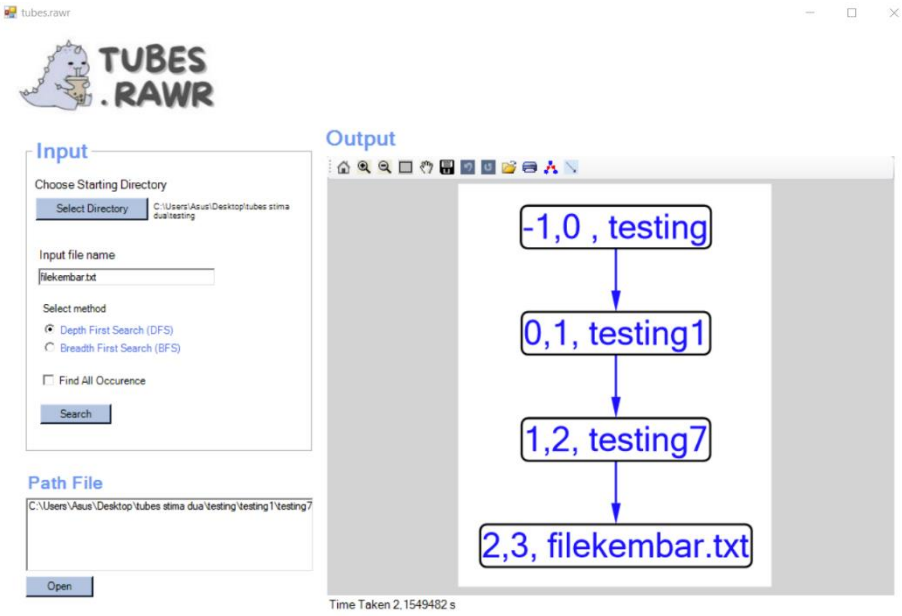
4.4.1 Pengujian Satu - BFS

Data uji yang pertama mengambil input dari direktori folder testing dengan nama file yang dicari adalah testing1. Simpul dan garis yang berwarna biru merupakan solusi untuk pencarian menggunakan algoritma BFS.



4.4.1 Pengujian dua - DFS

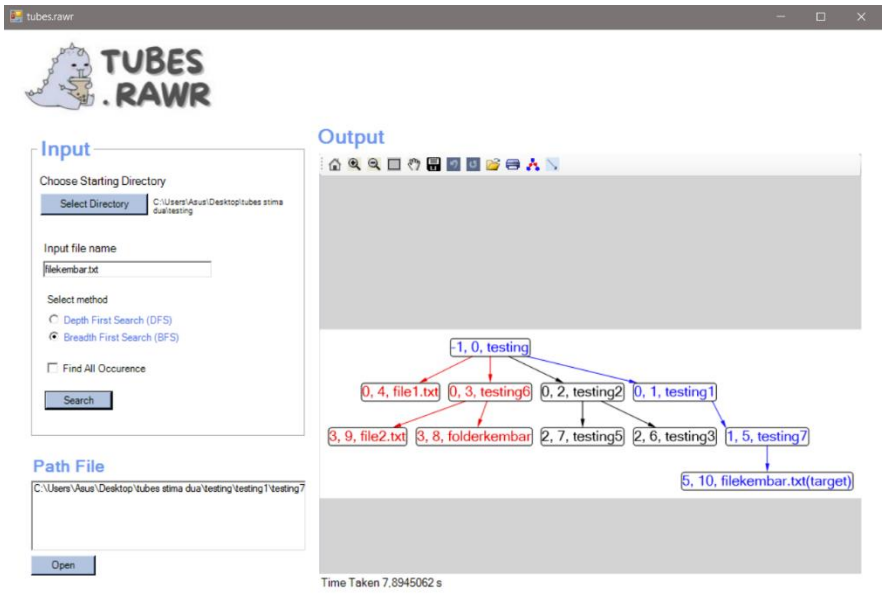
Pengujian kedua mengambil input dari direktori folder testing dengan nama file yang dicari adalah filekembar.txt. Simpul dan garis yang berwarna biru merupakan solusi untuk pencarian menggunakan algoritma DFS.



4.4.3. Pengujian Tiga - BFS

Pengujian ketiga mengambil input dari direktori folder testing dengan nama file yang dicari adalah **filekembar.txt**. Simpul dan garis yang berwarna biru merupakan solusi untuk pencarian menggunakan algoritma BFS. Sedangkan simpul dan garis berwarna merah

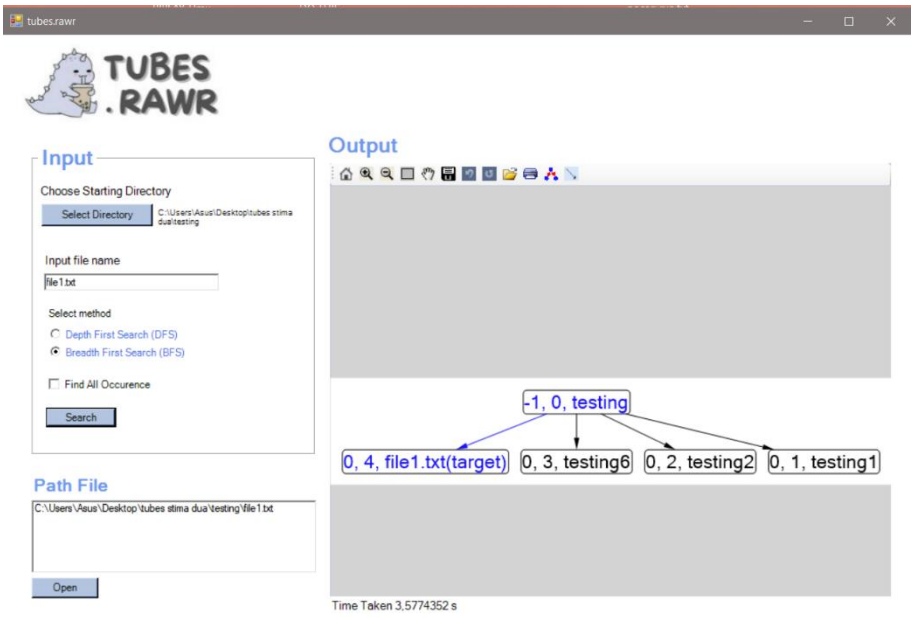
merupakan folder yang telah ditelusuri namun tidak ditemukan file tujuan, serta warna hitam merupakan folder yang belum/tidak ditelusuri oleh program.



Konfigurasi pengujian ini hampir sama dengan konfigurasi pengujian Dua, perbedaan terletak pada teknik. Pada pengujian dua, teknik yang digunakan adalah DFS. Sedangkan pengujian tiga menggunakan teknik DFS. Pengujian dua membutuhkan waktu 2 detik, sedangkan pengujian tiga membutuhkan waktu 7 detik. Pada kasus ini, DFS lebih cepat daripada BFS. Hal ini terjadi karena filekembar.txt terletak di dalam folder testing1, dimana pada kasus pencarian dengan DFS, pencarian dimulai dengan menelusuri folder testing1 secara mendalam, hingga seluruh simpulnya telah dikunjungi. Dengan demikian, filekembar.txt akan langsung ditemukan pada penelusuran folder pertama. Sedangkan pada kasus pencarian dengan BFS, program akan menelusuri terlebih dahulu setiap folder yang bertetangga, oleh karena itu, filekembar.txt akan ditemukan setelah program telah mengunjungi seluruh folder sebelumnya.

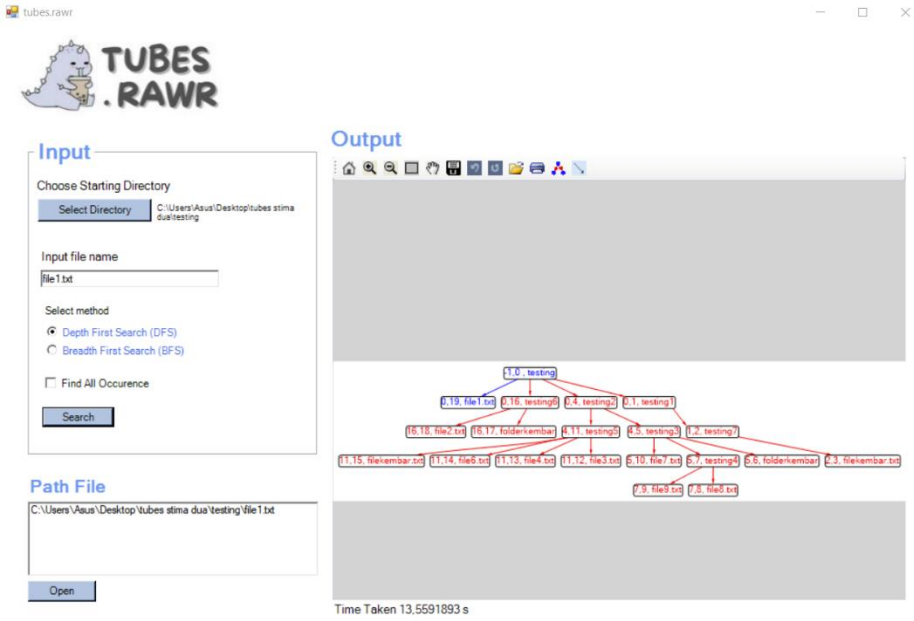
4.4.4 Pengujian Empat - BFS

Pengujian keempat mengambil input dari direktori folder testing dengan nama file yang dicari adalah file1.txt. Simpul dan garis yang berwarna biru merupakan solusi untuk pencarian menggunakan algoritma BFS.



4.4.5 Pengujian Lima - DFS

Pengujian kelima mengambil input dari direktori folder testing dengan nama file yang dicari adalah file1.txt. Simpul dan garis yang berwarna biru merupakan solusi untuk pencarian menggunakan algoritma BFS.



Konfigurasi pengujian lima hampir sama persis dengan pengujian Empat, perbedaan terletak pada teknik. Pengujian Empat menggunakan BFS, sedangkan pengujian Lima

menggunakan BFS. Pengujian empat membutuhkan 3 detik, sedangkan pengujian lima membutuhkan 13 detik. Pada kasus ini, BFS lebih cepat daripada DFS.

4.4.6 Analisis Implementasi Algoritma BFS dan DFS

Dari percobaan-percobaan di atas, dapat terlihat bahwa algoritma BFS lebih cepat bila *file* yang dicari berada pada kedalaman yang cukup dangkal. Dengan kata lain, posisi relatif *file* terhadap *root folder* tidak boleh terpisah terlalu jauh oleh *subdirectories*. Ini disebabkan algoritma BFS memeriksa semua simpul pada kedalaman *n* terlebih dahulu sebelum melanjutkan ke kedalaman *n+1*. Maka, semakin dekat posisi suatu simpul terhadap simpul awal atau semakin dangkal kedalaman suatu simpul pada sebuah graf, semakin cepat juga diperiksa simpul tersebut dibandingkan dengan simpul lainnya yang lebih jauh atau lebih dalam.

Sementara itu, algoritma DFS lebih cepat menemukan sebuah *file* apabila *file* tersebut berada jauh dari titik *folder* awal. Atau dengan kata lain, simpul *file* tersebut berada pada kedalaman mendekati kedalaman maksimum dari graf lengkapnya. Ini dikarenakan algoritma DFS akan mengecek hingga simpul dengan kedalaman maksimum baru perlahan-lahan melakukan "*backtracking*" simpul hingga semua simpul tetangga telah dikunjungi.

Namun, dari dua paragraf diatas, tidak berarti bahwa pada *file* yang berjarak jauh lebih baik dicari menggunakan algoritma DFS dan sebaliknya. Alasannya adalah implementasi algoritma BFS dan DFS pada traversal sebuah graf didasarkan pada fakta tidak adanya fungsi heuristik atau informasi tambahan lainnya (*blind search*). Meskipun demikian, dapat dipertimbangkan penggunaan algoritma BFS apabila direktori awal tidak memiliki banyak *subdirectory* dan algoritma DFS apabila sebaliknya yang berlaku.

BAB 5 – KESIMPULAN DAN SARAN

5.1 Kesimpulan

Terdapat beberapa metode untuk melakukan penjelajahan simpul dari representasi graf, diantaranya yaitu metode Breadth First Search (BFS) yaitu penelusuran secara melebar dan metode Depth First Search (DFS) yaitu penulusuran secara mendalam. Melalui tugas Folder Crawling dari mata kuliah IF2211 Strategi Algoritma ini, kami telah berhasil mengimplementasikan algoritma pencarian secara BFS dan DFS untuk menelusuri direktori folder dan mencari suatu file. Kelompok kami juga telah membuat aplikasi GUI sederhana untuk melakukan searching seperti fitur dari file explorer pada sistem operasi yang direpresentasikan ke dalam graf/pohon. User Interface dari aplikasi kami dibuat dengan framework .NET menggunakan bahasa pemrograman C#. Tidak ada yang secara mutlak lebih superior diantara kedua algoritma. Seberapa cepat penggunaan lebih bergantung terhadap kasus yang ingin diselesaikan, daripada terhadap algoritma yang digunakan.

5.2 Saran

Visualisasi graf menggunakan MSAGL dapat lebih dioptimalkan dengan mempelajarinya secara mendalam dengan waktu yang lebih lama. Dapat dipelajari lebih banyak tentang aplikasi desktop, bahasa pemrograman C#, serta framework .NET agar lebih familiar dan dapat mempersingkat waktu pengerjaan. Interface dari aplikasi juga dapat diperindah dengan menjadikannya responsif.

DAFTAR PUSTAKA

Munir, R. & Maulidevi, N. U. 2021. *Breadth/Depth First Search (BFS/DFS) Bagian 1*. Edunex. https://cdn-edunex.itb.ac.id/38015-Algorithm-Strategies-Parallel-Class/85259-BFS-dan-DFS/1645539911916_BFS-DFS-2021-Bag1.pdf

Munir, R. & Maulidevi, N. U. 2021. *Breadth/Depth First Search (BFS/DFS) Bagian 2*. Edunex. https://cdn-edunex.itb.ac.id/38015-Algorithm-Strategies-Parallel-Class/85259-BFS-dan-DFS/1647232725181_BFS-DFS-2021-Bag2.pdf

LINK PENTING

Link Github : https://github.com/LordGedelicious/Folder-Crawling-Implementation-with-BFS-DFS-Algorithm

Link Video YouTube Kelompok : https://youtu.be/PRud6zJoqwE