LAPORAN TUGAS BESAR 2 IF2211
STRATEGI ALGORITMA
Pengaplikasian Algoritma BFS dan DFS dalam Implementasi *Folder Crawling*

Dipersiapkan oleh :

WhyNotSearch - Kelompok 32

13520066 Putri Nurhaliza

13520140 Febryola Kurnia Putri

13520147 Aloysius Gilang Pramudya

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

2022

Daftar Isi

Daftar Gambar

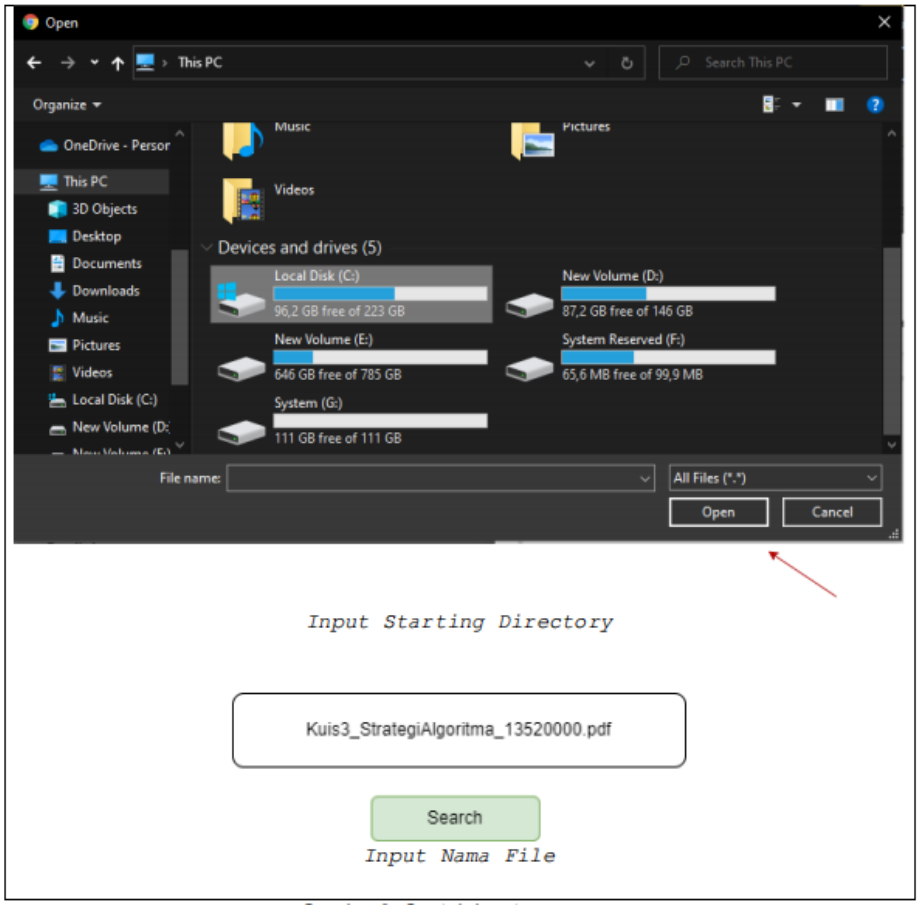
Daftar Tabel

Bab I
Deskripsi Tugas

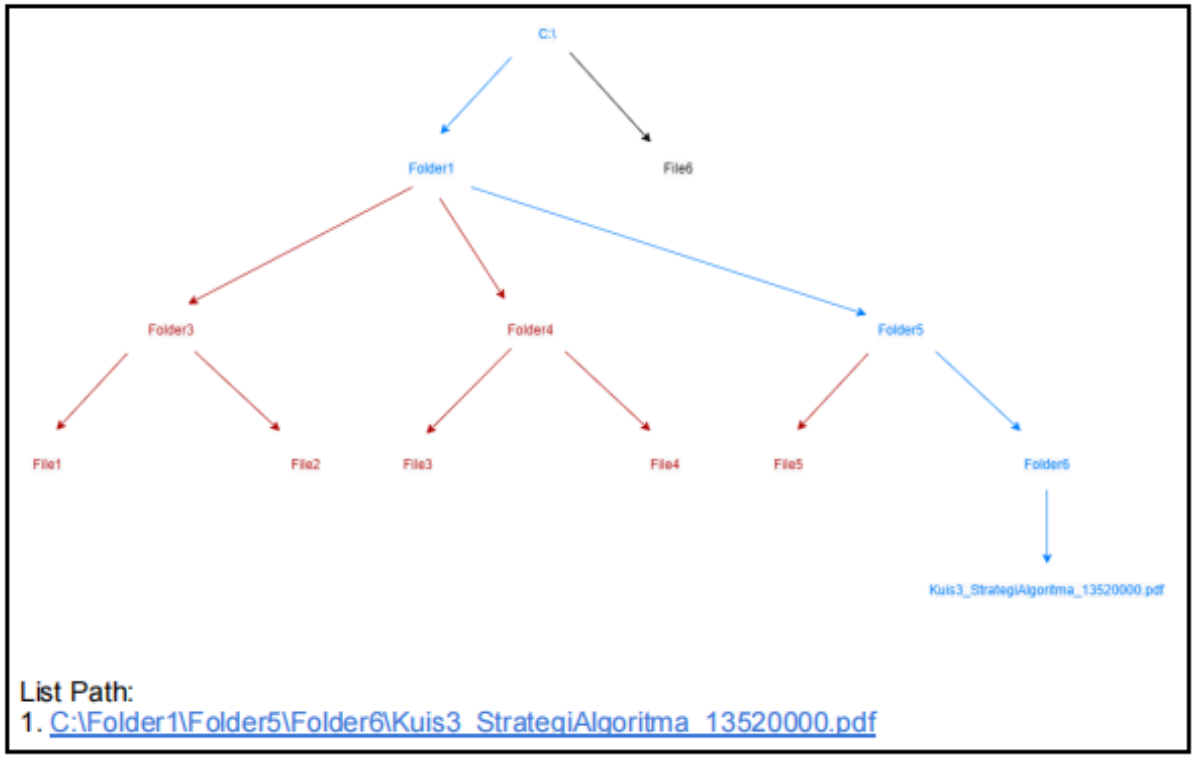
Dalam tugas besar ini, akan dibangun sebuah aplikasi GUI sederhana yang dapat memodelkan fitur dari *file explorer* pada sistem operasi, yang pada tugas ini disebut dengan *Folder Crawling*. Dengan memanfaatkan algoritma *Breadth First Search* (BFS) dan *Depth First Search* (DFS), maka dapat folder-folder 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:

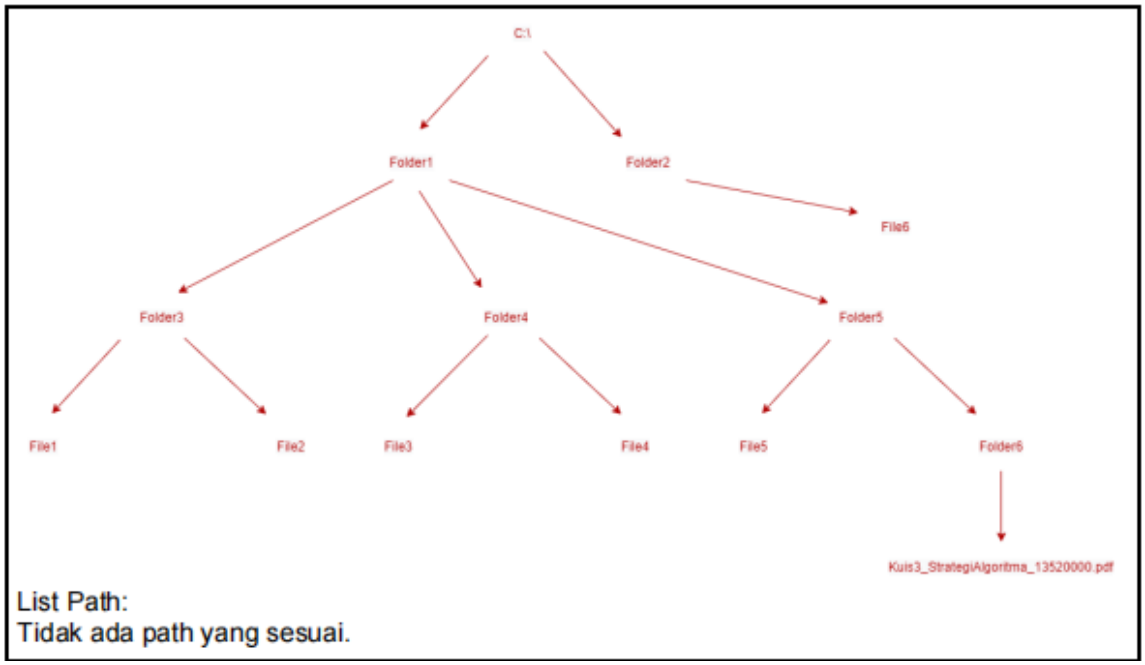


Gambar 1. Contoh input program



Gambar 2. Contoh output program

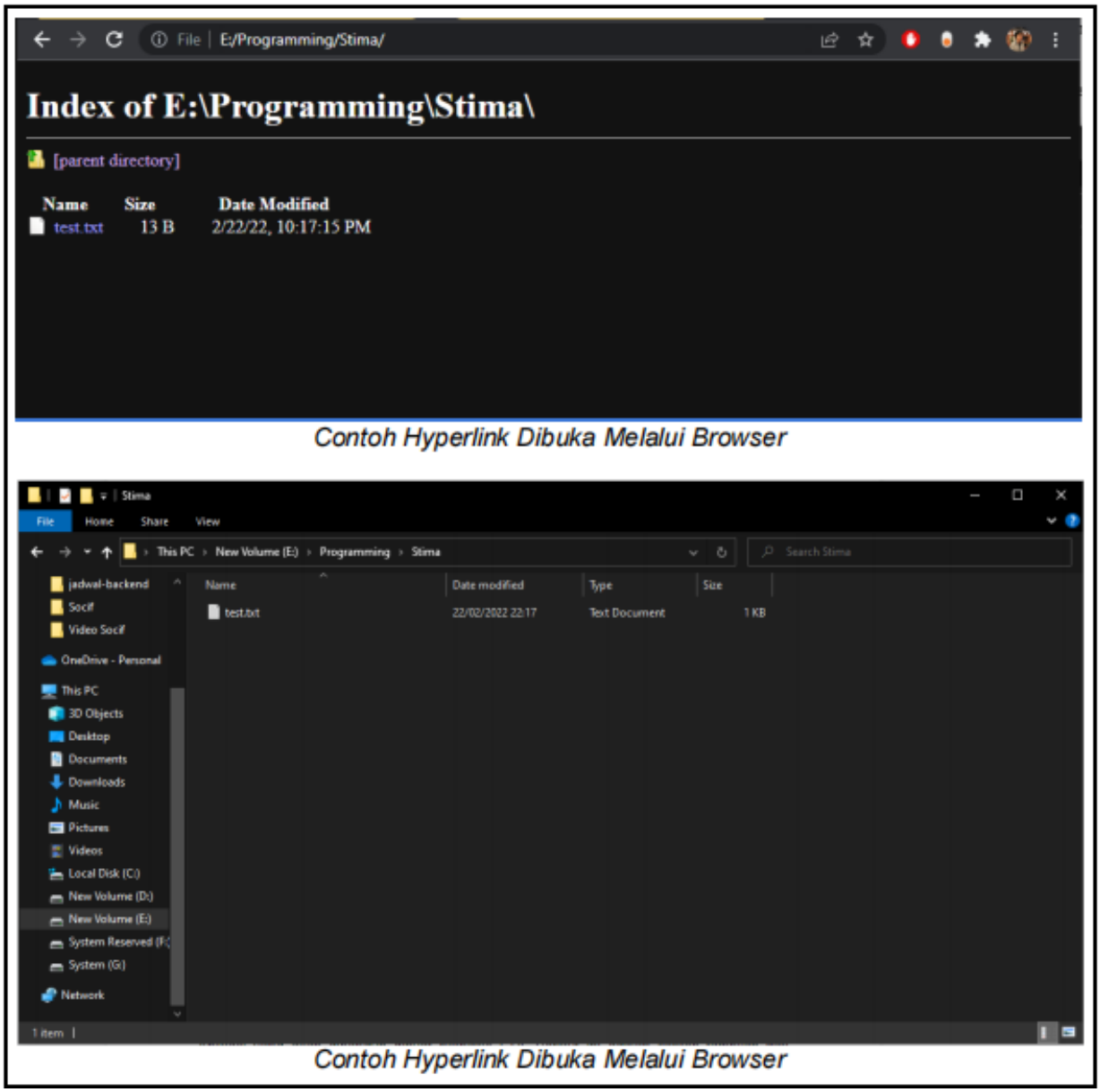
Misalnya pengguna ingin mengetahui langkah *folder crawling* untuk menemukan file Kuis3_StrategiAlgoritma_13520000.pdf. Maka, path pencarian DFS adalah sebagai berikut. C:\ → Folder1 → Folder3 → File1 → Folder3 → File2 → Folder3 → Folder1 → Folder4 → File3 → Folder4 → File4 → Folder4 → Folder1 → Folder5 → 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 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 4. Contoh ketika hyperlink di-klik

Bab II
Landasan Teori

2.1. Graf Traversal

Gambar 5. Contoh penerapan graf traversal

Graf traversal mengacu pada proses mengunjungi tiap simpul dalam graf, ada beberapa macam cara pengunjungan namun kali ini akan dibahas strategi BFS dan DFS. BFS atau yang kita sebut dengan Breadth-First Search adalah suatu strategi yang mengunjungi semua simpul tetangga suatu simpul sebelum mengunjungi anak simpul tersebut, proses ini cenderung lebih lama dibandingkan DFS namun dijamin menemukan solusinya. Berkebalikan dengan BFS, DFS atau Depth-First Search adalah suatu strategi algoritma yang mengunjungi semua anak suatu simpul sebelum mengunjungi simpul tetangga, proses ini memang memiliki waktu lebih cepat namun tidak dijamin menemukan solusi. Algoritma pencarian solusi pada graf traversal, terbagi menjadi dua, yaitu:

1. **Tanpa informasi (uninformed/blind search)**
   • Tidak ada informasi tambahan
   • Contoh: DFS, BFS, Depth Limited Search, Iterative Deepening Search.
2. **Dengan informasi (informed Search)**
   • Pencarian berbasis heuristik
   • Mengetahui non-goal state “lebih menjanjikan” daripada yang lain
   • Contoh: Best First Search, A

2.2. Algoritma BFS

Algoritma BFS (Breadth-First Search) merupakan salah satu algoritma yang digunakan dalam pencarian graf. Konsep algoritma BFS adalah melakukan pencarian untuk node saat ini, lalu melakukan pencarian untuk semua node tetangga yang belum pernah dikunjungi secara berurutan. Algoritma ini menyimpan daftar node tetangga yang akan dikunjungi dengan menggunakan struktur data queue. Proses kerja algoritma BFS adalah sebagai berikut:

1. Masukkan node akar ke queue.
2. Keluarkan node yang ada di head dari queue. Jika node yang dikeluarkan adalah solusi dari persoalan, berhenti dan kembalikan node tersebut. Jika tidak, masukkan semua node tetangga yang belum pernah diperiksa ke dalam queue.
3. Jika queue kosong, maka semua node sudah diperiksa. Hentikan pencarian dan kembalikan nilai null.
4. Jika queue tidak kosong, ulangi langkah 2.

Berikut adalah pseudocode dari algoritma BFS.

```
BFS(G,s)
        for each vertex u ∈ V [G] − {s} do
                state[u] = “undiscovered”
                p[u] = nil, i.e. no parent is in the BFS tree
        state[s] = “discovered”
        p[s] = nil
        Q = {s}
        while Q != Ø do
                u = dequeue[Q]
                process vertex u as desired
                for each v ∈ Adj[u] do
                        ocess edge (u,v) as desired
                        if state[v] = “undiscovered” then
                        state[v] = “discovered”
                        p[v] = u
                        enqueue[Q,v]
                state[u] = “processed”
```

Gambar 6. Pseudocode Algoritma BFS

2.3. Algoritma DFS

Algoritma DFS (Depth-First Search) merupakan salah satu algoritma yang digunakan dalam pencarian graf. Konsep algoritma DFS adalah melakukan pencarian melakukan pencarian pada suatu node, lalu melakukan pencarian pada node pertama yang belum pernah dikunjungi hingga solusi ditemukan atau tidak ada node tetangga yang belum dikunjungi. Algoritma ini menyimpan daftar node tetangga yang akan dikunjungi dengan menggunakan struktur data stack.

Dari sini dapat dilihat bahwa algoritma DFS merupakan bentuk khusus dari algoritma backtracking. Proses kerja algoritma DFS adalah sebagai berikut:

1. Masukkan node akar ke stack.
2. Keluarkan node yang ada di top dari stack. Jika node yang dikeluarkan adalah solusi dari persoalan, berhenti dan kembalikan node tersebut. Jika tidak, masukkan semua node tetangga yang belum pernah diperiksa ke dalam stack.
3. Jika stack kosong, maka semua node sudah diperiksa. Hentikan pencarian dan kembalikan nilai null.
4. Jika stack tidak kosong, ulangi langkah 2.

Proses kerja algoritma DFS di atas adalah algoritma DFS yang dilakukan secara iteratif.

Berikut merupakan *pseudocode* dari algoritma DFS yang dilakukan secara iteratif.

```
procedure DFS-iterative(G,v):
        label v as discovered
        let S be a stack
        S.push(v)
        while S is not empty
                t ← S.peek()
                if t is what we're looking for:
                        return t
                for all edges e in G.adjacentEdges(t) do
                        if edge e is already labelled
                                continue with the next edge
                        w ← G.adjacentVertex(t,e)
                        if vertex w is not discovered and not explored
                                label e as tree-edge
                                label w as discovered
                                S.push(w)
                                continue
                        else if vertex w is discovered
                                label e as back-edge
                        else
                                // vertex w is explored
                                label e as forward- or cross-edge
                label t as explored
                S.pop()
```

Gambar 7. Pseudocode Algoritma DFS secara iteratif

Berikut merupakan *pseudocode* dari algoritma DFS yang dilakukan secara rekursif.

```
DFS(G,u)
        state[u] = "discovered"
        process vertex u if desired
        entry[u] = time
        time = time + 1
        for each v ∈ Adj[u] do
        process edge (u,v) if desired
        if state[v] = "undiscovered" then
                p[v] = u
                DFS(G,v)
        state[u] = "processed"
        exit[u] = time
        time = time + 1
```

Gambar 8. Pseudocode Algoritma DFS secara rekursif

2.4. Penjelasan Singkat C#

Desktop app ini ditulis dengan bahasa C# (C-Sharp) di dalam aplikasi visual studio, dengan menggunakan windows Form. Windows Form atau sering disebut dengan WinForm adalah Class library buatan microsoft yang bersifat GUI atau Graphical User Interface. Windows Form tergabung dalam Framework .Net. Tujuan diciptakan Win Form adalah untuk mempermudah developer dalam membuat suatu aplikasi berbasis desktop. Bayak sekali fitur yang disediakan oleh winform, seperti label, panel dan beberapa fitur lainya yang menunjang developer dalam UI/UX aplikasi.

Bab III
Analisis Pemecahan Masalah

3.1. Langkah-Langkah Pemecahan Masalah

Pada persoalan ini proses pencarian dilakukan menggunakan algoritma BFS dan DFS. Pendekatan graf dinamis digunakan pada persoalan ini karena graf belum tersedia sebelum pencarian dan akan dibangun selama pencarian solusi. Langkah-langkah yang dilakukan adalah sebagai berikut.

A. Pembangunan pohon dimulai dari simpul *starting directory*
B. Simpul di ekspan berdasarkan folder/file yang dimilikinya
C. Graf ditelusuri berdasarkan aturan BFS atau DFS
D. Setiap simpul diperiksa apakah solusi (goal) telah dicapai atau tidak.
E. Jika ya, maka pencarian selesai (jika hanya ingin mencari satu solusi saja) atau dilanjutkan mencari solusi lain (jika ingin mencari semua solusi).
F. Jika tidak, maka pencarian dilanjutkan ke simpul berikutnya berdasarkan aturan BFS/DFS

3.2. Proses Mapping Persoalan

Pada proses pencarian solusi, kelompok kami menggunakan pendekatan graf dinamis. Graf tidak akan tersedia sebelum pencarian, graf akan dibangun selama pencarian solusi. representasi pohon dinamis pada persoalan ini adalah sebagai berikut.

A. Pohon ruang status (*state space tree)*
   Pohon ruang status merupakan pohon yang berisi semua simpul yang dibentuk selama pencarian solusi. Pada persoalan ini kami merepresentasikan pohon ruang status dengan Adjacency List yang bertipe data kamus(*dictionary)* dengan *key* merepresentasikan simpul dan *value* merepresentasikan simpul tetangganya.

B. Simpul
   Pada persoalan ini simpul merepresentasikan nama *file/folder* pada direktori pencarian. Simpul akar adalah direktori awal mula pencarian sedangkan simpul daun merupakan *goal state*

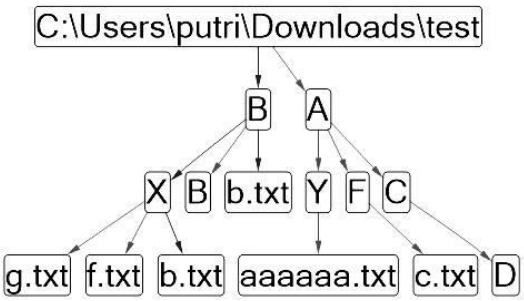
C. Cabang
   Pada setiap percabangan akan dicek apabila nama *file/folder* sudah sesuai dengan target yang dicari. Jika pencarian hanya membutuhkan satu hasil saja maka pencarian akan dihentikan. Namun jika pencarian menerima semua hasil yang sesuai maka pencarian akan diteruskan sampai pembentukan pohon selesai.

D. Ruang status (*state space)*
Pada persoalan ini ruang status merupakan himpunan semua simpul yaitu semua *file/folder* yang ada pada direktori pencarian.

E. Ruang solusi
Ruang solusi menampung semua simpul solusi dari akar ke *goal state.*

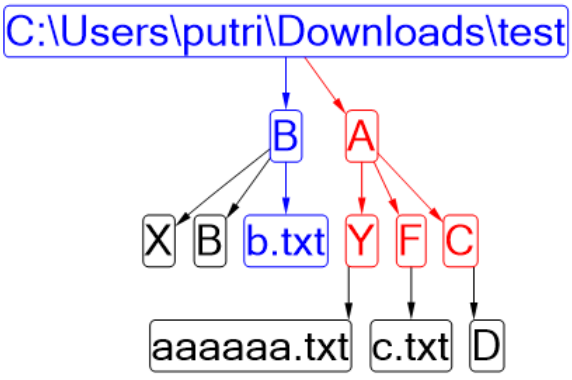
3.3. Ilustrasi Kasus Lain

Contoh pada spesifikasi tugas menggambarkan kasus penggunaan DFS untuk mencari 1 file yang sesuai. Berikut adalah ilustrasi kasus lain yang berhasil kami temukan dan selesaikan. Contoh berikut menggunakan pohon dibawah ini.



Gambar 9. Kasus lain

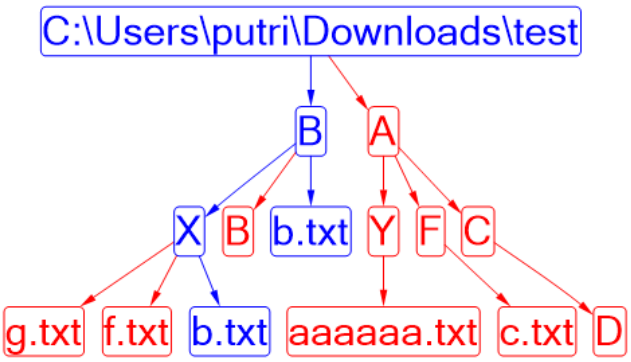
A. Pencarian 1 file menggunakan BFS



Gambar 10. Kasus lain 1

Mencari b.txt,
C:\Users\putri\Downloads\test -> A -> B -> C:\Users\putri\Downloads\ -> A -> C -> A -> F -> A -> Y -> A -> C:\Users\putri\Downloads\ -> B-> b.txt

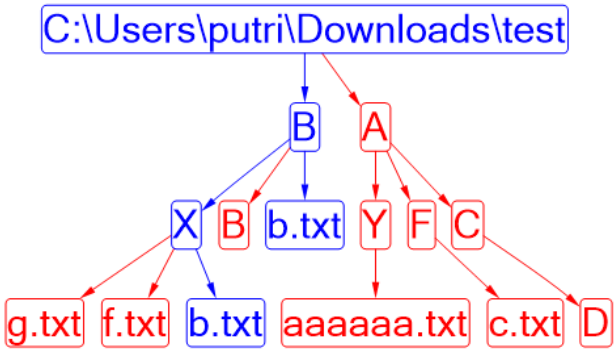
B. Pencarian semua file yang sesuai menggunakan BFS



Gambar 11. Kasus lain 2

Mencari b.txt,
C:\Users\putri\Downloads\test -> A -> B -> C:\Users\putri\Downloads\ -> A -> C -> A -> F -> A -> Y -> A -> C:\Users\putri\Downloads\ -> B-> b.txt -> B -> B -> B -> X -> B -> C:\Users\putri\Downloads\test -> A -> C -> D -> C -> A -> F -> c.txt -> F -> A -> Y -> aaaaaa.txt -> Y -> A -> C:\Users\putri\Downloads\ -> B -> X -> b.txt -> X -> f.txt -> X -> g.txt

C. Pencarian semua file yang sesuai menggunakan DFS

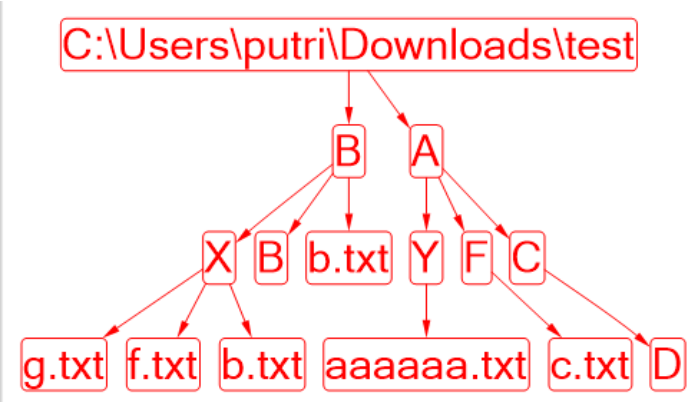


Gambar 12. Kasus lain 3

C:\Users\putri\Downloads\test -> B -> X -> g.txt -> X -> f.txt -> X -> b.txt -> X -> B -> B -> B -> b.txt -> B -> C:\Users\putri\Downloads\test -> -> Y -> aaaaaa.txt -> Y -> A -> F -> c.txt -> F -> A -> C -> D

(keterangan: karena DFS diimplementasikan menggunakan stack, maka simpul yang dipilih untuk di telusuri lebih dulu terurut terbalik secara alfabet (folder dahulu baru file). Sebagai contoh, dari *starting directory*, B ditelusuri lebih dahulu daripada A. Sederhananya, berdasarkan gambar diatas pemeriksaan dilakukan dari simpul terkiri dari simpul-simpul dengan level dan parent yang sama).

D. Pencarian file dan tidak ditemukan (menggunakan BFS/DFS)



Gambar 13. Kasus lain 4

Akan menelusuri seluruh simpul sesuai aturan BFS/DFS.

E. Pencarian file di starting directory yang kosong (menggunakan BFS/DFS)
Tidak terbentuk pohon karena hanya ada simpul *starting directory*.

Bab IV
Implementasi dan Pengujian

4.1. Implementasi Program

Tabel 4.1 Pseudocode Program

```
//Algoritma BFS dan DFS
//Program dibuat dengan membuat class Graph sebagai kelas utama
//dan kelas BFS dan DFS sebagai turunannya
//Bagian pertama akan dijelaskan pseudocode mengenai kode pada kelas Graph
class Graph
class BFS inherit Graph
class DFS inherit Graph
Deklarasi
totalNodes = int
totalEdges = int
file = string
dir = string
adjacencyList = list of string
parentAndChildren = list of string
visited = set of string
visitedVertex = set of string
predPath = list of string
predVertex = list of string
returnVertex = set of string
returnMultipleVertex = set of string

function main(){
 input(startingdirectory)
 input(file)
 if(!buttonCheckAll){
   if(Algoritma=BFS){
   resultpath = singleSearchBFS(startingdirectory,file)
   addKeyPath(directory, "");
    returnPath = directory;
    lastVertex = directory;
    this.file = file;
    visited.Add(directory);
    visitedVertex.Add(directory);
    // iterasi pertama
    searchQueue.Enqueue(directory);
    //pencarian dilakukan selama masih ada node dalam queue
    while (searchQueue.Count != 0)
      {
      vertex = searchQueue.Dequeue();      // didequeue untuk diperiksa
      _vertex = " ";
      if (Equals(vertex, directory))
      {
        _vertex = directory;
      }
      else
       {
        _vertex = DirectoryInfo(vertex).Name;
```

```
         visited.Add(vertex);
         visitedVertex.Add(_vertex);
         ParentAndChildren.Add(Tuple.Create(predPath[vertex], vertex));
         if (Equals(_vertex, file))
         {
          // file yang sesuai ditemukan
          getPath(directory, vertex);
          ParentAndChildren.Add(Tuple.Create(predPath[vertex], vertex));
          return vertex;
          }
         }
        if (!Directory.Exists(vertex))
            {
                continue;
            }
        files = Directory.GetFiles(vertex);
        dirs = Directory.GetDirectories(vertex);
        allfiles = List of files;
        allfiles.AddRange(files);
        allfiles.AddRange(dirs);

        foreach (s in allfiles)
            {
            predPath.Add(s, vertex);
            _s = DirectoryInfo(s).Name;
            if (!visited.Contains(s))
            {
              addVertex(_s);

              if (string.Equals(vertex, directory))
              {
                 addEdge(Tuple.Create(vertex, _s));
              }
              else
              {
                 addEdge(Tuple.Create(_vertex, _s));
              }
              ParentAndChildren.Add(Tuple.Create(vertex, s));

              totalEdges++;
              totalNodes++;

              searchQueue.Enqueue(s);

             }
             else { continue; }

             }

      }

      return returnPath;
     }
    }
    else{
       addKeyPath(directory, "");
       returnPath = directory;
```

```
        lastVertex = directory;
        this.file = file;
         visited.Add(directory);
         visitedVertex.Add(directory);

         // iterasi pertama
         searchStack.Push(directory);

         // pencarian dilakukan selama masih ada node dalam stack
         while (searchStack.Count != 0)
         {
             vertex = searchStack.Pop();          // dipop untuk diperiksa
             _vertex = " ";
             if (Equals(vertex, directory))
             {
                 _vertex = directory;
             }
             else
             {
                 _vertex = new DirectoryInfo(vertex).Name;
                 visited.Add(vertex);
                 visitedVertex.Add(_vertex);
                 ParentAndChildren.Add(Tuple.Create(predPath[vertex], vertex));
                 if (Equals(_vertex, file))
                 {
                     // file yang sesuai ditemukan
                     getPath(directory, vertex);
                     ParentAndChildren.Add(Tuple.Create(predPath[vertex],
vertex));
                     return vertex;

                 }
             }
             if (!Directory.Exists(vertex))
             {
                 continue;
             }
             files = Directory.GetFiles(vertex);
             dirs = Directory.GetDirectories(vertex);
             List<string> allfiles = List<string>();
             allfiles.AddRange(files);
             allfiles.AddRange(dirs);
             visited.Add(vertex);
             visitedVertex.Add(_vertex);
             foreach (s in allfiles)
             {
                 predPath.Add(s, vertex);
                 _s = new DirectoryInfo(s).Name;

                 if (!visited.Contains(s))
                 {
                     this.addVertex(_s);

                     if (string.Equals(vertex, directory))
                     {
                         addEdge(Tuple.Create(vertex, _s));
                     }
                     else
                     {
                         addEdge(Tuple.Create(_vertex, _s));
```

```
                    }
                    ParentAndChildren.Add(Tuple.Create(vertex, s));

                    totalEdges++; totalNodes++;

                    searchStack.Push(s);
                }
                else { continue; }

            }
        }
        return returnPath;
        }
 }
 else{
   if(Algoritma=BFS)
   {
         addKeyPath(directory, "");
        List<string> returnPath = List<string>();
        file = file;
        visited.Add(directory);
        visitedVertex.Add(directory);

        // iterasi pertama
        searchQueue.Enqueue(directory);

        // pencarian dilakukan selama masih ada node dalam queue
        while (searchQueue.Count != 0)
        {
            vertex = searchQueue.Dequeue();              // didequeue untuk
diperiksa
            _vertex = " ";
            if (Equals(vertex, directory))
            {
                _vertex = directory;
            }
            else
            {
                _vertex = new DirectoryInfo(vertex).Name;
                visited.Add(vertex);
                visitedVertex.Add(_vertex);
                ParentAndChildren.Add(Tuple.Create(predPath[vertex], vertex));

                if (Equals(_vertex, file))
                {
                    //
                    getMultiplePath(directory, vertex);
                    ParentAndChildren.Add(Tuple.Create(predPath[vertex],
vertex));
                    returnPath.Add(vertex);
                }
            }
            if (!Directory.Exists(vertex))
            {
                continue;
            }
            files = Directory.GetFiles(vertex);
            dirs = Directory.GetDirectories(vertex);
            allfiles = List<string>();
```

```
            allfiles.AddRange(files);
            allfiles.AddRange(dirs);
            foreach (s in allfiles)
            {
                predPath.Add(s, vertex);
                _s = DirectoryInfo(s).Name;

                if (!visited.Contains(s))
                {
                    addVertex(_s);

                    if (Equals(vertex, directory))
                    {
                        addEdge(Tuple.Create(vertex, _s));
                    }
                    else
                    {
                        addEdge(Tuple.Create(_vertex, _s));
                    }

                    ParentAndChildren.Add(Tuple.Create(vertex, s));

                    totalEdges++;
                    totalNodes++;

                    searchQueue.Enqueue(s);
                }
                else { continue; }

            }

        }

        return returnPath;
    }

   else{
         addKeyPath(directory, "");
        returnPath = List<string>();
        file = file;
        visited.Add(directory);
        visitedVertex.Add(directory);

        // iterasi pertama
        searchStack.Push(directory);

        // pencarian dilakukan selama masih ada node dalam stack
        while (searchStack.Count != 0)
        {
            vertex = searchStack.Pop();              // dipop untuk diperiksa
            _vertex = " ";
            if (Equals(vertex, directory))
            {
                _vertex = directory;
            }
            else
            {
                _vertex = DirectoryInfo(vertex).Name;
```

```
                ParentAndChildren.Add(Tuple.Create(predPath[vertex], vertex));

                if (Equals(_vertex, file))
                {
                    // file yang sesuai ditemukan
                    getMultiplePath(directory, vertex);
                   ParentAndChildren.Add(Tuple.Create(predPath[vertex],
vertex));
                    returnPath.Add(vertex);
                }
            }
            if (!Directory.Exists(vertex))
            {
                continue;
            }
            files = Directory.GetFiles(vertex);
            dirs = Directory.GetDirectories(vertex);
            allfiles = List of file;
            allfiles.AddRange(files);
            allfiles.AddRange(dirs);
            visited.Add(vertex);
            visitedVertex.Add(_vertex);
            foreach (s in allfiles)
            {
                predPath.Add(s, vertex);
                _s = DirectoryInfo(s).Name;

                if (!visited.Contains(s))
                {
                    this.addVertex(_s);

                    if (Equals(vertex, directory))
                    {
                        addEdge(Tuple.Create(vertex, _s));
                    }
                    else
                    {
                        addEdge(Tuple.Create(_vertex, _s));
                    }

                    ParentAndChildren.Add(Tuple.Create(vertex, s));

                    totalEdges++; totalNodes++;

                    searchStack.Push(s);

                }
                else { continue; }
            }
        }
        return returnPath;
    }
   }
}
```

4.2. Struktur Data

A. Class Form1

Class yang merupakan kerangka aplikasi. Atributnya adalah seluruh objek yang ada pada tampilan dan dapat befungsi sebagai input atau output. Terdiri atas label, radio button, panel, text box, check box, rich text box, trackbar, dan print dialog. Method pada Form 1 berikut secara umum mengatur event yang terjadi pada program.

public Form1()	Konstruktor untuk menginisialisasi semua komponen
private void buttonRefresh_Click(object sender, EventArgs e)	Menginisialisasi ulang semua komponen
private void buttonChooseFolder_Click(object sender, EventArgs e)	Memilih folder sebagai *root directory*
private void buttonSearch_Click(object sender, EventArgs e)	Menggunakan class BFS ataupun DFS untuk melakukan pencarian (satu atau semua) berdasarkan kondisi pada radioButton dan checkBox. Menghasilkan graf yang terdefenisi sesuai pencarian yang telah dilakukan
private void drawGraph(Graph graf)	Menggunakan MSAGL graph untuk menampilkan pertumbuhan pohon
private void bindGraph (Microsoft.Msagl.Drawing.Graph graph)	Memasukkan MSAGL graph pada panel
public void wait(int milliseconds)	Menambahkan delay dalam program dengan tujuan pertumbuhan pohon dapat dilihat sesuai kecepatan yang diinginkan
public void visitedFolderSingle(string folder)	Menambahkan path hasil pencarian yangs sesuai (hanya 1) untuk ditampilkan ke result
public void visitedFolderMultiple(string folder)	Menambahkan path hasil pencarian yang sesuai (semua) untuk ditampilkan ke result
private void Link_LinkClicked1(object sender, LinkLabelLinkClickedEventArgs e)	Membuka file explorer berdasarkan *result path* yang diklik

B. Class Graph

1. Data
   a. private int totalNodes

Menyimpan total simpul pada graf

b. private int totalEdges
   Menyimpan total edge pada graf
c. private string file
   Menyimpan nama file yang akan dicari
d. private string dir
   Meyimpan *root* pencarian
e. private Dictionary<string, HashSet<string>> AdjacencyList
   List yang menyimpan ketetanggan antar simpul. Misal parameter pertama adalah simpul A, maka parameter kedua adalah semua simpul yang bertetangga dengan simpul A
f. private List<Tuple<string, string>> ParentAndChildren
   List pasangan parent dan child. Parameter pertama adalah parent, parameter kedua adalah child.
g. public HashSet<string> visited
   Menyimpan simpul yang telah dikunjungi (dalam bentuk *full path*)
h. public HashSet<string> visitedVertex
   Menyimpan simpul yang telah dikunjungi (hanya nama)
i. public Dictionary<string, string> predPath
   Parameter pertama (*key*) adalah path yang sedang diperiksa, parameter kedua (*value*) adalah path sebelumnya/parent.
j. public Dictionary<string, string> predVertex
   Parameter pertama (*key*) adalah simpul yang sedang diperiksa, parameter kedua (*value*) adalah simpul sebelumnya/parent.
k. public HashSet<string> returnVertex
   Menyimpan hasil pencarian (hanya 1)
l. public HashSet<HashSet<string>> returnMultipleVertex
   Menyimpan hasil pencarian (semua)

2. Method

public Graph()	default konstruktor
public Graph(string dir, string file)	konstruktor dengan parameter dir dan file yang akan dicari
public string File	set = mengubah file menjadi value get = return nama file
public string Dir	set = mengubah dir menjadi value get = return dir
public List<Tuple<string, string>> getParentAndChildren()	return ParentAndChildren
public void addVertex(string vertex)	menambahkan simpul baru ke adjacencyList dengan tetangganya masih kosong
public void addEdge(Tuple<string, string> edge)	menambahkan tetangga ke simpul yang sudah ada

public void getPath(string root, string file)	mendapatkan path hasil pencarian (hanya 1)
public void getMultiplePath(string root, string file)	mendapatkan path hasil pencarian (semua)
public void addKeyPath(string key, string path)	menambahkan key dan value baru ke predPath

C. Class BFS

Class BFS inheritance ke class graph

1. Data
   a. public Queue<string> searchQueue
      Antrian (*queue*) simpul-simpul yang dibangkitkan dan diperiksa
2. Method

public BFS(Graph graf)	konstruktor dengan graf terdefenisi
public BFS(string root, string file)	konstruktor dengan parameter dir dan file yang akan dicari
public string singleSearchBFS(string directory, string file)	fungsi utama untuk pencarian 1 file yang sesuai secara BFS
public List<string> multipleSearchBFS(string directory, string file)	fungsi utama untuk pencarian semua file yang sesuai secara BFS

D. Class DFS

Class DFS inheritance ke class graph

1. Data
   a. public Queue<string> searchStack
      Tumpukan (*stack*) simpul-simpul yang dibangkitkan dan diperiksa
2. Method

public DFS(Graph graf)	konstruktor dengan graf terdefenisi
public DFS(string root, string file)	konstruktor dengan parameter dir dan file yang akan dicari
public string singleSearchDFS(string directory, string file)	fungsi utama untuk pencarian 1 file yang sesuai secara DFS
public List<string> multipleSearchDFS(string directory, string file)	fungsi utama untuk pencarian semua file yang sesuai secara DFS

4.3. Tata Cara Penggunaan Program

A. Setup

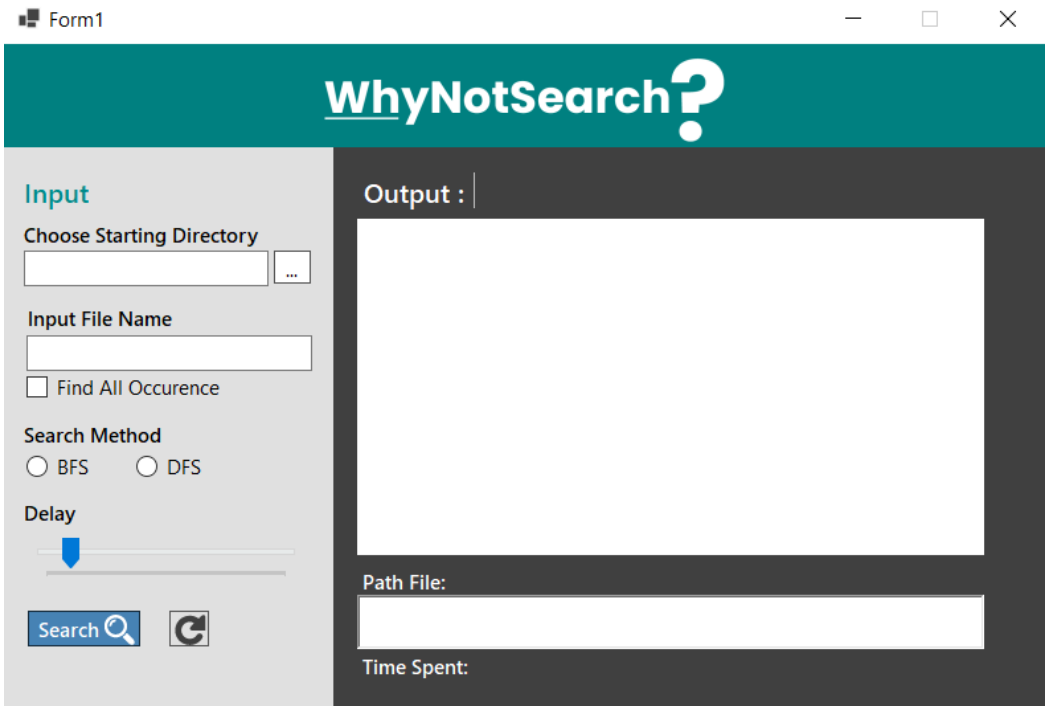
1. Install Visual Studio 2022 atau versi sebelumnya
2. Create project Desktop app

B. Penggunaan Program

1. Jalankan executable code (src.exe) pada folder bin
2. Browse dan pilih file/folder yang ingin dilakukan pencarian
3. Jika hanya ingin mencari satu file/folder saja, maka abaikan button “Find All Occurence”
4. Jika ingin mencari lebih dari satu file/folder, maka klik button “Find All Occurance”
5. Pilih Metode pencarian (BFS atau DFS)
6. Atur Delay penggambaran graf yang diinginkan
7. Klik tombol search
8. Kumpulan path file/folder yang dicari akan masuk ke box path dan jika ingin di klik akan membuka direktori tempat file/folder tersebut berada
9. Jika ingin melakukan pencarian ulang, klik button refresh yang berada di sebelah tombol search
10. Lakukan ulang seperti Langkah 1-8

4.4. Hasil Pengujian

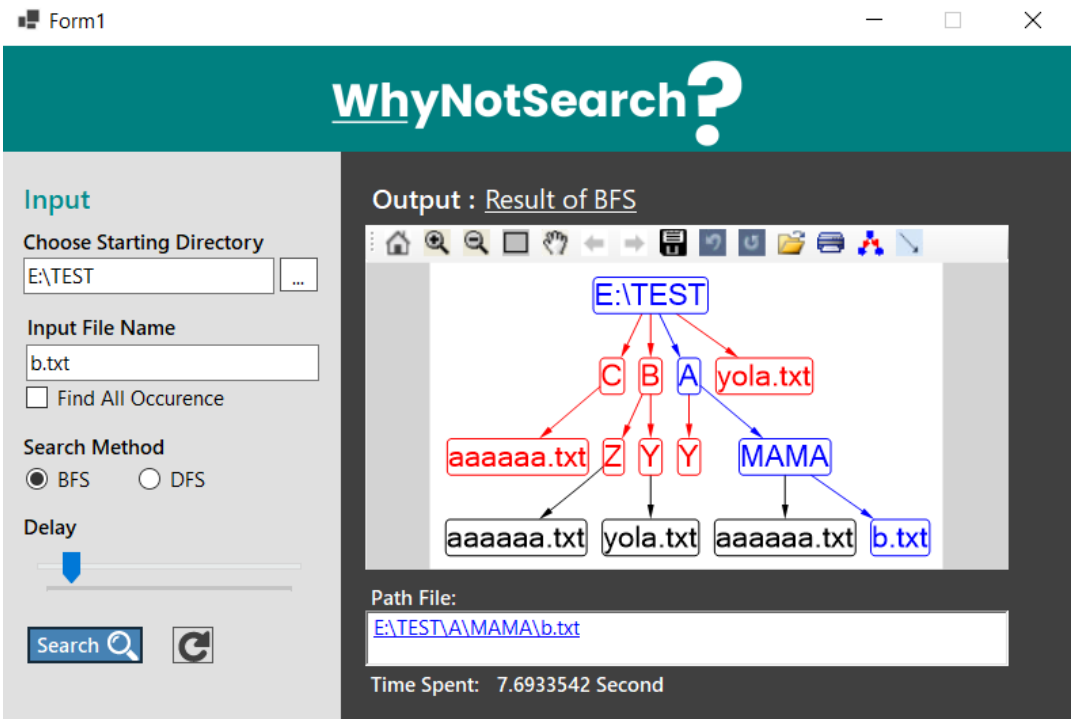
- **Interface Awal Program**



Gambar 14. Interface awal program

- **Pengujian 1**

  Case:
  1. Mencari file
  2. File ada di direktori
  3. Pencarian dengan single search (hanya perlu menemukan satu file)
  4. Method dengan BFS

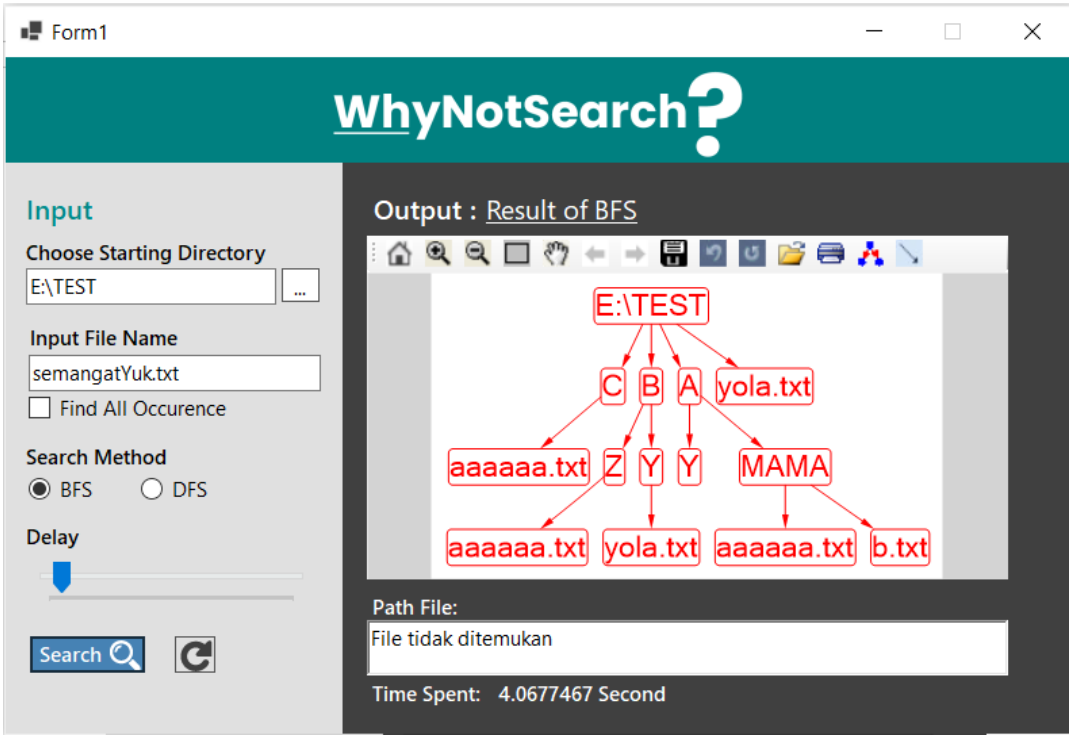


Gambar 15. Pengujian 1

- **Pengujian 2**

  Case:

  1. Mencari file
  2. File tidak ada di direktori
  3. Pencarian dengan single search (hanya perlu menemukan satu file)
  4. Method dengan BFS

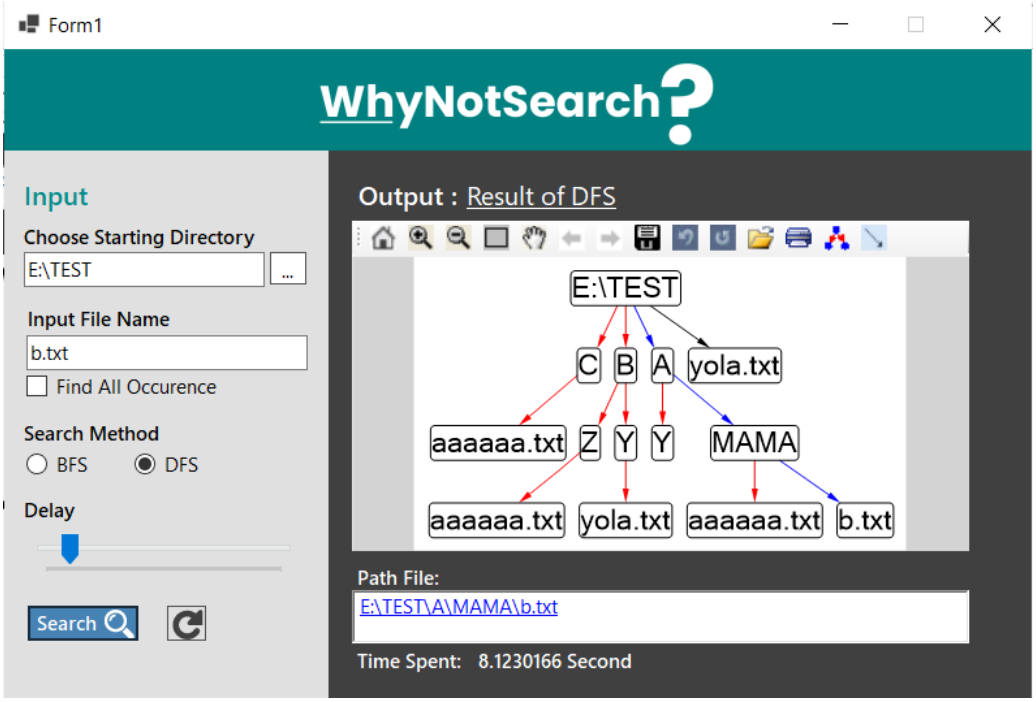


Gambar 16. Pengujian 2

- **Pengujian 3**

  Case:

  1. Mencari file
  2. File ada di direktori
  3. Pencarian dengan single search (hanya perlu menemukan satu file)
  4. Method dengan DFS

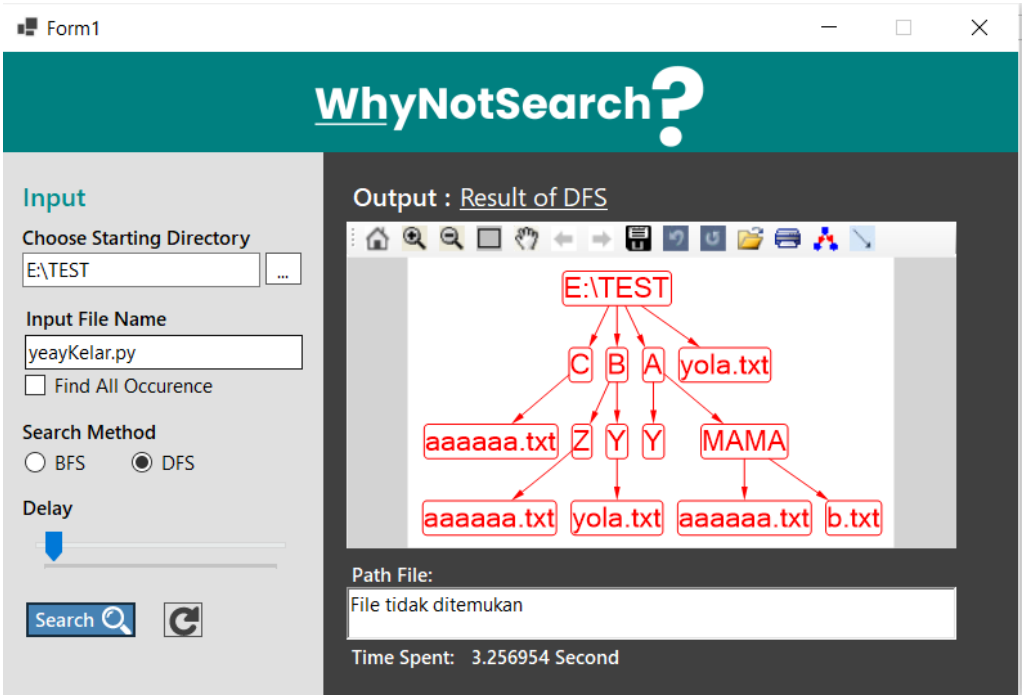


Gambar 17. Pengujian 3

- **Pengujian 4**

  Case:

  1. Mencari file
  2. File tidak ada di direktori
  3. Pencarian dengan single search (hanya perlu menemukan satu file)
  4. Method dengan DFS

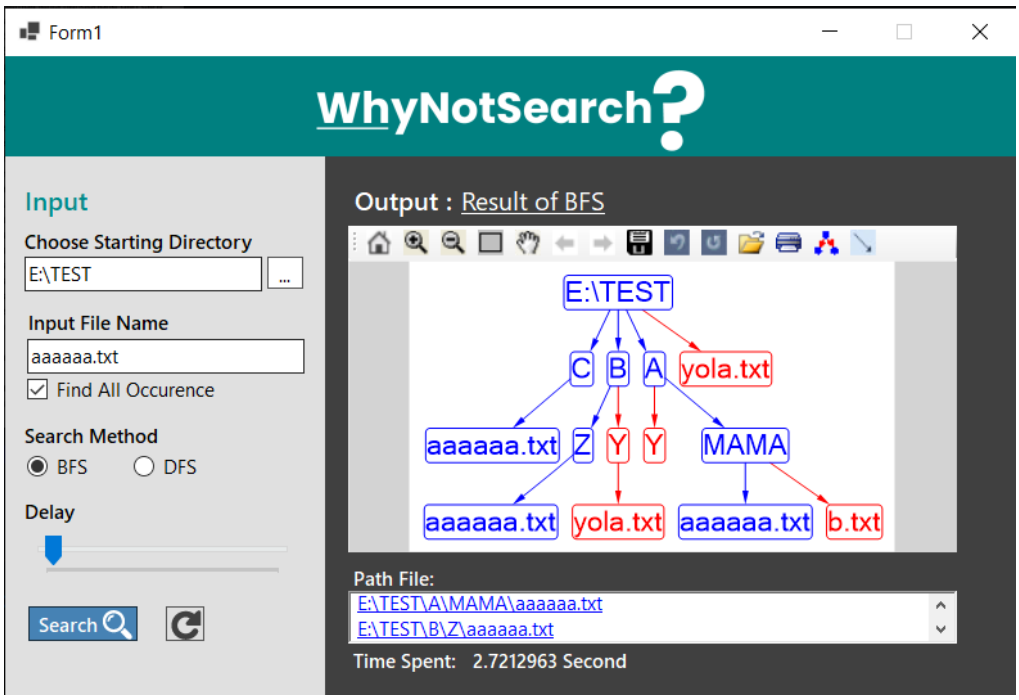


Gambar 18. Pengujian 4

- **Pengujian 5**

  Case:

  1. Mencari file
  2. File ada di direktori
  3. Pencarian dengan multiple search (menemukan lebih dari 1 file)
  4. Method dengan BFS

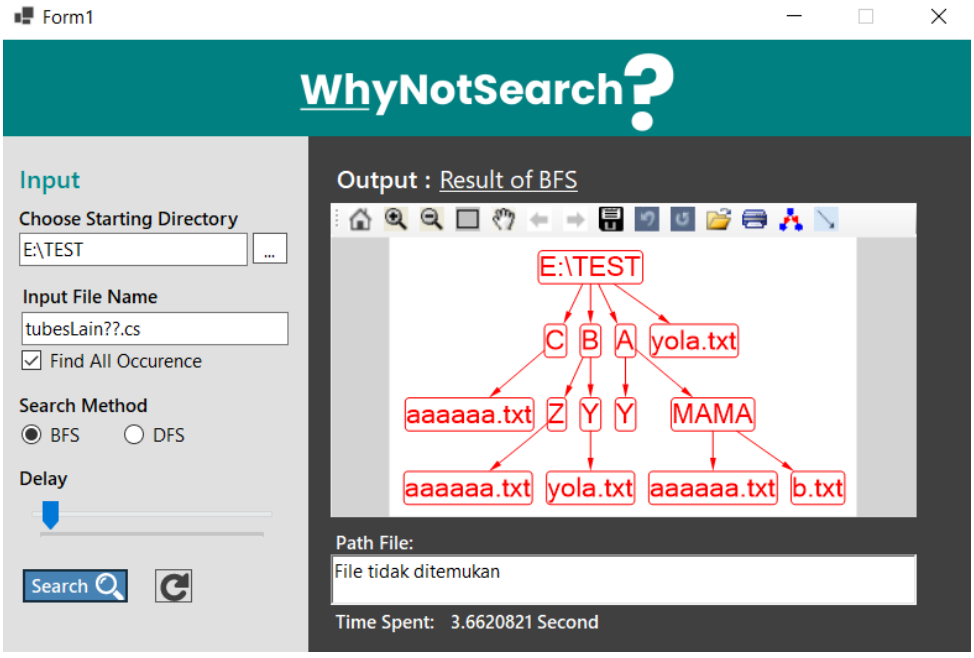


Gambar 19. Pengujian 5

- **Pengujian 6**

  Case:

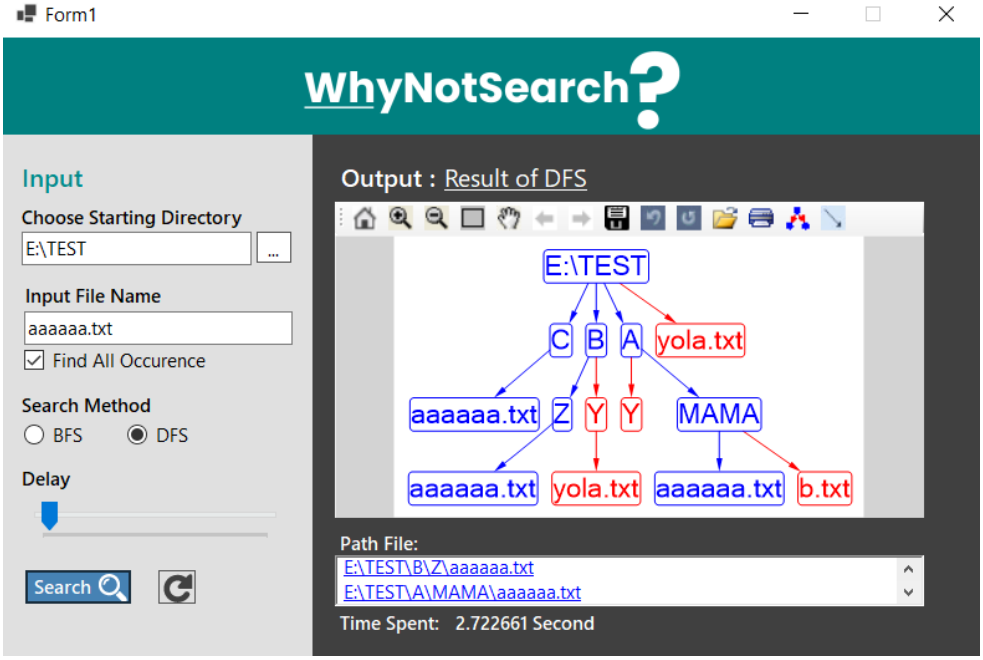
  1. Mencari file
  2. File tidak ada di direktori
  3. Pencarian dengan multiple search (menemukan lebih dari 1 file)
  4. Method dengan BFS



Gambar 20. Pengujian 6

- **Pengujian 7**

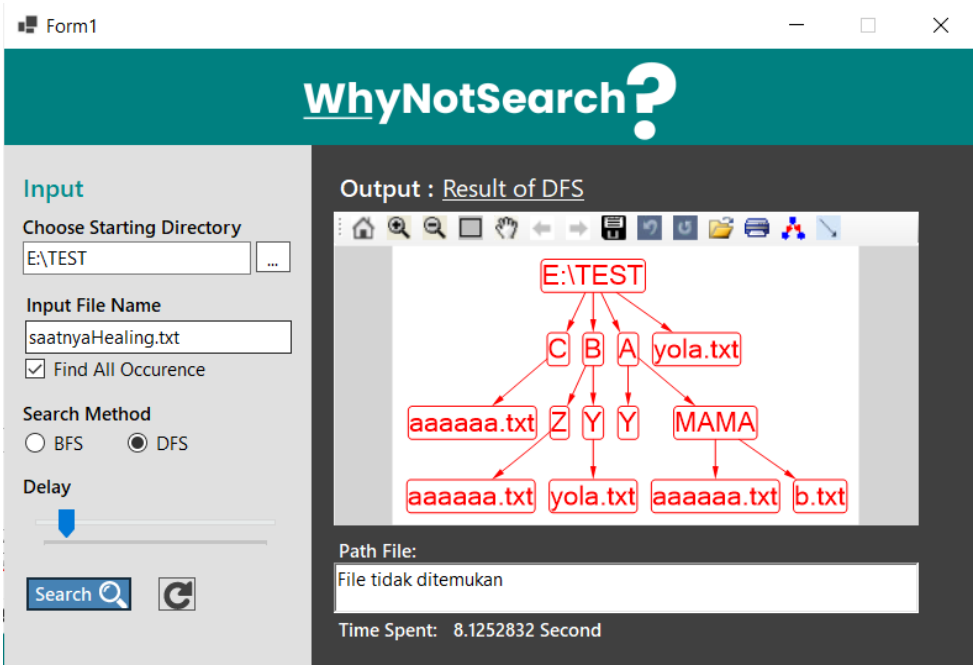
  Case:
  1. Mencari file
  2. File ada di direktori
  3. Pencarian dengan multiple search (menemukan lebih dari 1 file)
  4. Method dengan DFS



Gambar 21. Pengujian 7

- **Pengujian 8**

  Case:
  1. Mencari file
  2. File tidak ada di direktori
  3. Pencarian dengan multiple search (menemukan lebih dari 1 file)
  4. Method dengan DFS

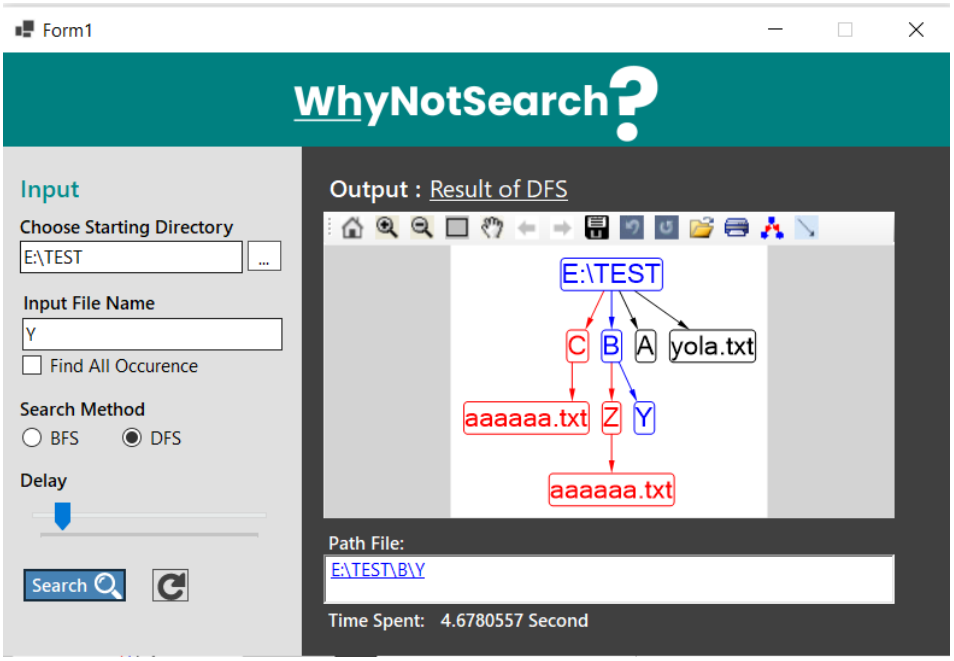


Gambar 22. Pengujian 8

- **Pengujian 9**

  Case:

  1. Mencari folder
  2. folder ada di direktori
  3. Pencarian dengan single search (hanya perlu menemukan satu folder)
  4. Method dengan DFS



Gambar 23. Pengujian 9

4.5. Analisis Design Solusi BFS dan DFS

Berdasarkan pengujian yang telah dilakukan, ditemukan bahwa BFS maupun DFS memiliki performa terbaik dalam kasus tertentu. BFS memberikan hasil yang lebih baik jika file yang dicari berada di level kedalaman yang lebih kecil (dekat dengan *starting directory*) dengan cabang yang tidak terlalu banyak. Artinya, kita tidak perlu menelusuri lebih dalam. Sementara itu, DFS memberikan hasil yang lebih baik jika file yang dicari ada di simpul awal yang di ekspan, sehingga kita tidak perlu menelusuri cabang lain. Kedua metode memberikan hasil yang sama jika file tidak ditemukan dan jika pencarian dilakukan untuk menemukan seluruh file yang sesuai. Hal ini terjadi karena baik BFS maupun DFS harus menelusuri seluruh simpul berdasarkan aturannya masing-masing.

Bab V
Kesimpulan dan Saran

5.1. Kesimpulan

Dari tugas besar IF2211 Strategi Algoritma semester 2 2021/2022 berjudul "Pengaplikasian Algoritma BFS dan DFS dalam Implementasi *Folder Crawling*", kami berhasil membuat sebuah *form berbasis windows* untuk mencari path file dan membentuk pohon pencarian path file yang dibuat menggunakan kakas .NET dengan mengaplikasikan algoritma pencarian BFS dan DFS. Program berhasil dibuat dengan baik dan tanpa error dan sesuai dengan spesifikasi Tugas Besar 2 IF2211 Strategi Algoritma yang telah diberikan.

5.2. Saran

Saran-saran yang dapat kami berikan untuk tugas besar IF2211 Strategi Algoritma semester 2 2021/2022 adalah:

1. Algoritma yang *dig*unakan pada Tugas Besar ini masih memiliki banyak kekurangan sehingga sangat memungkinkan untuk dilakukan efisiensi, misalnya dengan tidak menggunakan fungsi yang sama berulang-ulang. Oleh karena itu, dalam pengembangan program ini, masih bisa dilakukan efisiensi kinerja.
2. Penulisan *pseudocode* tampak kurang perlu dikarenakan program yang lumayan panjang dan membaca program lebih mudah daripada membaca *pseudocode* dengan asumsi program sudah *well commented.*
3. Memperjelas pemberian spesifikasi dan batasan-batasan setiap program pada *file* tugas besar untuk mencegah adanya multitafsir dan kesalahpahaman pada proses pembuatan program.

Lampiran

Link Video :

Link Repository Github:

Daftar Pustaka

Slide kuliah IF2211 Strategi Algoritma tahun ajaran 2021/2022.