

**LAPORAN TUGAS BESAR 2
IF2211 STRATEGI ALGORITMA**

**PENGAPLIKASIAN ALGORITMA BFS
DAN DFS DALAM IMPLEMENTASI
*FOLDER CRAWLING***



Dipersiapkan oleh:

Kelompok DBFS (23)

Ilham Pratama	13520041
Eiffel Aqila Amarendra	13520074
Raka Wirabuana Ninagan	13520134

**Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2022**

DAFTAR ISI

DAFTAR ISI	<i>i</i>
BAB I	1
BAB II	4
Graph Traversal	4
Breadth First Search (BFS)	4
Depth First Search (DFS)	4
Penjelasan Mengenai C# Desktop Application Development	5
BAB III	6
Langkah-Langkah Pemecahan Masalah	6
Proses Mapping Persoalan Menjadi Elemen-Elemen Algoritma BFS dan DFS	6
Contoh Ilustrasi Kasus Lain	7
BAB IV	9
Implementasi Program	9
Struktur Data	16
Tata Cara Penggunaan Program	16
Hasil Pengujian	18
Analisis Desain Solusi Algoritma BFS dan DFS berdasarkan Hasil Pengujian	21
BAB V	23
Kesimpulan	23
Saran	23
DAFTAR PUSTAKA	24
LAMPIRAN	25

BAB I

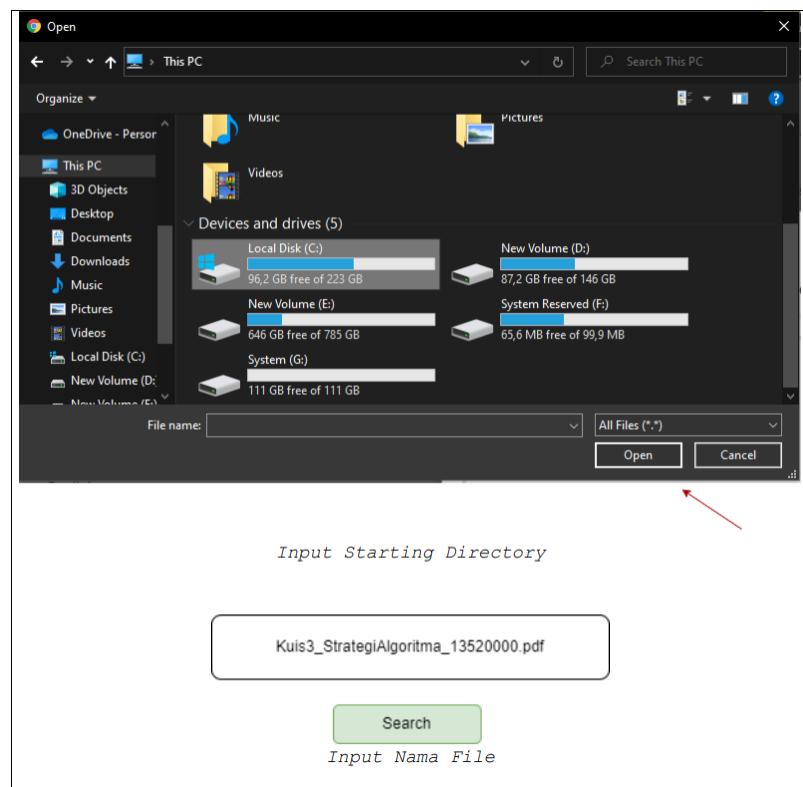
DESKRIPSI TUGAS

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

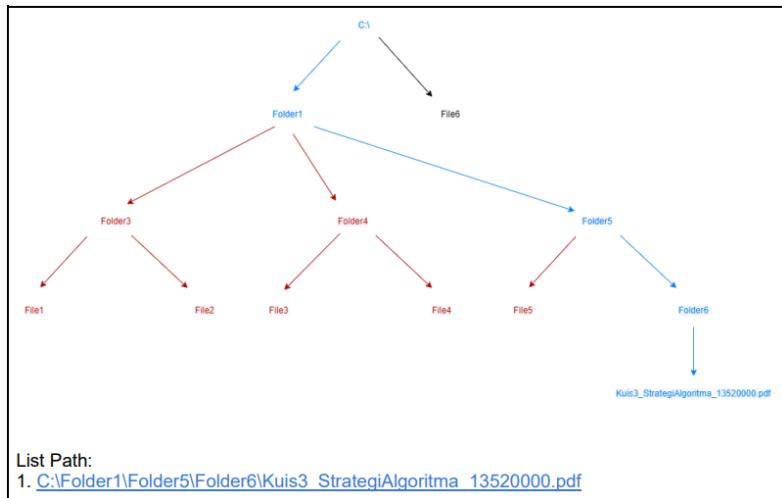
Selain pohon, Anda diminta juga menampilkan list path dari daun-daun yang bersesuaian dengan hasil pencarian. Path tersebut diharuskan memiliki hyperlink menuju folder parent dari file yang dicari, agar file langsung dapat diakses melalui browser atau file explorer. Contoh hal-hal yang dimaksud akan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

Contoh Input dan Output Program

Contoh masukan aplikasi:

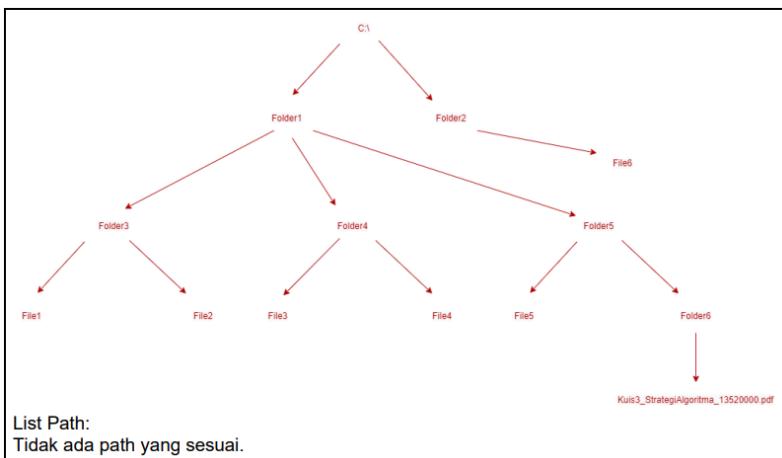


Contoh output aplikasi:



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.

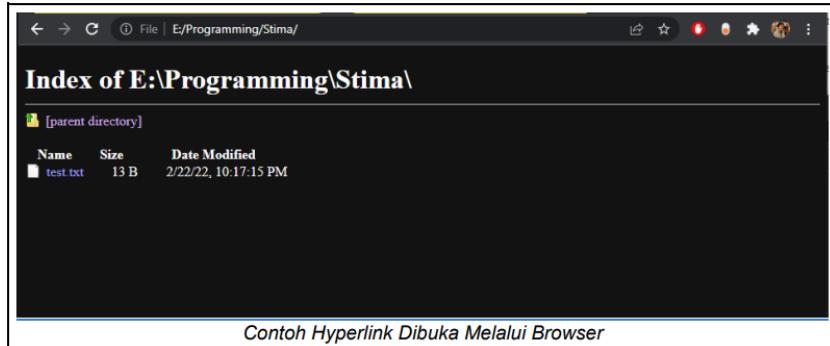
Pada gambar di atas, rute yang dilewati pada pencarian DFS diwarnai dengan warna merah. Sedangkan, rute untuk menuju tempat file berada diberi warna biru. Rute yang masuk antrian tapi belum diperiksa diberi warna hitam. Anda bebas menentukan warnanya asalkan dibedakan antara ketiga hal tersebut.



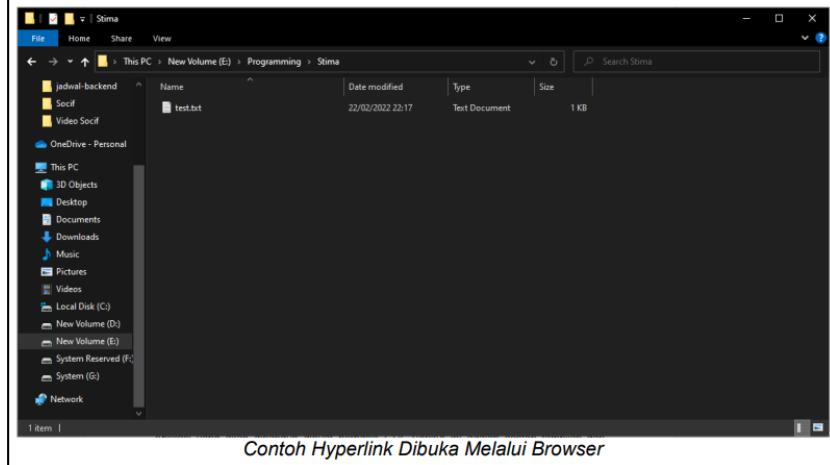
Jika file yang ingin dicari pengguna tidak ada pada direktori file, misalnya saat pengguna mencari Kuis3Probstat.pdf, maka path pencarian DFS adalah sebagai berikut: C:\ → Folder1 → Folder3 → File1 → Folder3 → File2 → Folder3 → Folder1 → Folder4 → File3 → Folder4 → File4 → Folder4 → Folder1 → Folder5 → 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:



Contoh Hyperlink Dibuka Melalui Browser



Contoh Hyperlink Dibuka Melalui Browser

BAB II

LANDASAN TEORI

2.1. Graph Traversal

Graph Traversal merupakan suatu algoritma penelusuran pada graph yang dilakukan dengan cara mengunjungi setiap simpul atau node yang ada pada graph secara sistematis. *Graph traversal* terdiri dari 2 metode yaitu *Breadth First Search* (BFS) dan *Depth First Search* (DFS). Terdapat dua pendekatan representasi *graph* dalam proses pencarian yaitu graf statis dan graf dinamis. Graf statis adalah graf yang sudah terbentuk sebelum proses pencarian dilakukan, sedangkan graf dinamis adalah graf yang terbentuk saat proses pencarian dilakukan.

2.2. Breadth First Search (BFS)

Breadth First Search (BFS) atau Pencarian Melebar merupakan algoritma graf traversal dengan mengunjungi sebuah simpul, kemudian melanjutkan kunjungan terhadap seluruh simpul yang bertetangga dengan simpul tersebut. Setelah memeriksa seluruh simpul tetangga, pencarian dilanjutkan terhadap simpul tetangga tersebut yang belum dikunjungi, demikian seterusnya.

2.3. Depth First Search (DFS)

Depth First Search (BFS) atau Pencarian Mendalam merupakan algoritma graf traversal dengan mengunjungi sebuah simpul, kemudian melanjutkan kunjungan terhadap sebuah simpul yang bertetangga dengan simpul tersebut, demikian seterusnya. Setelah mencapai suatu simpul sedemikian sehingga seluruh simpul yang bertetangga dengannya telah dikunjungi, pencarian runut-balik (*backtrack*) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul tetangga yang belum dikunjungi. Pencarian berakhir ketika tidak ada simpul yang belum atau dapat dikunjungi dari simpul yang telah dikunjungi.

2.4. Penjelasan Mengenai C# Desktop Application Development

C# atau yang dibaca “C sharp” adalah bahasa pemrograman sederhana yang digunakan untuk tujuan umum. Dengan kata lain, bahasa pemrograman ini dapat digunakan untuk berbagai fungsi, seperti pemrograman *server-side* pada *website*, membangun aplikasi baik *Desktop Application* maupun *Mobile Application*, pemrograman *game*, dan sebagainya. Bahasa C# sangat bergantung dengan framework yang disebut .NET Framework yang digunakan untuk mengcompile dan menjalankan kode C# dan sebuah IDE, yakni Visual Studio.

Visual Studio menyediakan berbagai jenis *project template* dalam berbagai bahasa pemrograman, platform, serta tipe. Untuk pengembangan *Desktop Application* dengan bahasa pemrograman C#, Visual Studio menyediakan berbagai *template* yang dapat dimanfaatkan pemrogram untuk membangun aplikasinya. Dalam Tugas Besar II IF2211 Strategi Algoritma ini, kelompok kami menggunakan “Windows Forms App (.NET Framework) sebagai *project template* awal kami.

BAB III

ANALISIS PEMECAHAN MASALAH

3.1. Langkah-Langkah Pemecahan Masalah

Dalam proses pemecahan permasalahan pengaplikasian algoritma BFS dan DFS dalam implementasi *folder crawling* ini, kelompok kami melakukan langkah-langkah sebagai berikut. Pertama-tama, kelompok kami memahami terlebih dahulu permasalahan yang tertera pada Bab I Deskripsi Tugas. Langkah berikutnya, kami mempelajari terlebih dahulu konsep, teknik, dan kakas yang dibutuhkan dalam implementasi program, seperti mempelajari konsep pemrosesan *folder* dan *file*, pengimplementasian algoritma BFS dan DFS dalam bahasa pemrograman C#, penggunaan kakas Visual Studio dan framework .NET, dan visualisasi *graph*. Kemudian, kami mencari referensi dari internet mengenai projek yang memiliki kemiripan dengan permasalahan ini, dan memahami struktur dari referensi tersebut.

Setelah itu, kelompok kami melakukan sebuah pengujian secara mikro dengan melakukan penulisan kode kasar program dalam 3 kategori, yaitu pengaplikasian GUI, pemrosesan *folder* dan *file* menjadi graf, dan pengaplikasian algoritma DFS dan BFS. Langkah selanjutnya, kelompok kami mencoba berbagai struktur penulisan kode pada pemrosesan *folder* dan *file* sampai ditemukannya struktur yang paling nyaman untuk digunakan dan dipahami untuk proses pemrosesan pencarian selanjutnya karena kode di bagian ini sangat terikat dengan algoritma pencarian dan memiliki beberapa keterikatan dengan GUI. Terakhir, kami mengimplementasikan algoritma DFS dan BFS pada program, memeriksa, memperbaiki, dan mengembangkan fitur dan program, serta melakukan proses *debugging* di masing-masing kategori dan program.

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

Elemen-elemen algoritma BFS dan DFS dipetakan dari permasalahan menjadi seperti berikut ini:

1. Algoritma BFS dan DFS merupakan algoritma pemrosesan struktur data graf, sehingga dibentuk 3 buah larik, antara lain
 - a. Larik yang mencatat seluruh *nodes*

- b. Larik yang mencatat seluruh *parent nodes*
 - c. Larik yang mencatat seluruh *child nodes*
2. Pada algoritma DFS, pencarian dilakukan secara rekursif sehingga akan muncul elemen pencatat rute pencarian saat rekurens dieksekusi.
 3. Pada BFS, pencarian dilakukan secara berurutan berdasarkan kunjungan ke semua *child* terlebih dahulu, sehingga digunakan *queue* untuk mengatur *nodes* prioritas yang perlu dieksekusi duluan.

3.3. Contoh Ilustrasi Kasus Lain

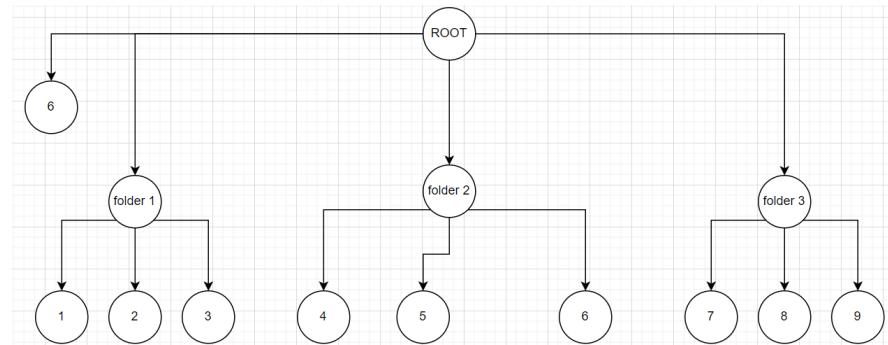
Contoh struktur dari sebuah direktori:

```

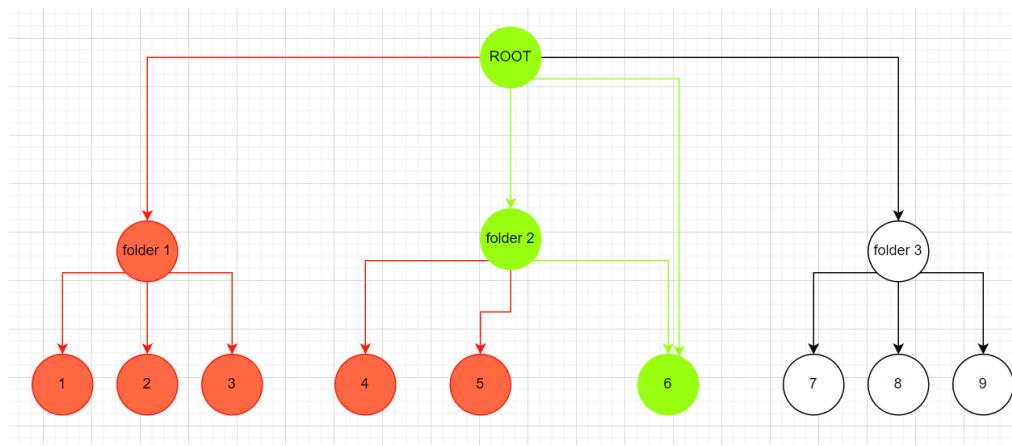
root/
└── folder 1/
    ├── 1
    ├── 2
    └── 3
└── folder 2/
    ├── 4
    ├── 5
    └── 6
└── folder 3/
    ├── 7
    ├── 8
    └── 9
└── 6

```

Pada ilustrasi berikut, seorang pengguna ingin mencari suatu *file* dengan nama “6” dengan *starting directory* ‘direktori awal’ pencarian adalah root. Berikut ditampilkan sebuah *tree* dari ilustrasi tersebut.



Pada ilustrasi tersebut, dapat terlihat bahwa terdapat 2 buah *file* dengan nama “6” pada direktori tersebut, dengan salah satu *file* berada pada *folder* root dan yang lainnya berada pada root/folder 2. Jika dilakukan proses pencarian, hasil yang akan diperoleh oleh program adalah bahwa program akan menunjukkan bahwa *file* dengan nama “6” akan disajikan sebagai 1 node, bukan 2 node yang terpisah. Berikut merupakan contoh tampilan, ketika dilakukan pencarian dengan menggunakan algoritma DFS (*Depth First Search*).



BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1. Implementasi Program

Program *Folder Crawling* kami terbagi menjadi empat buah program utama, antara lain Form1.cs, FolderCrawler.cs, BFS.cs, dan DFS.cs. Program Form1.cs memiliki fungsi sebagai GUI dengan menyediakan berbagai objek, seperti TextBox, Label, Button, CheckBox, ComboBox, dan GViewer. Program FolderCrawler.cs memiliki fungsi sebagai pemroses masukkan pengguna pada *form*, pemroses *folder* dan *file* menjadi graf, dan penghubung antara Form.cs dengan BFS.cs atau DFS.cs. Program BFS.cs memiliki fungsi sebagai pencari nama *file* masukkan pengguna dengan menggunakan algoritma BFS. Sedangkan, program DFS.cs memiliki fungsi sebagai pencari nama *file* masukkan pengguna dengan menggunakan algoritma DFS.

Program kami memiliki fitur utama, yakni pencarian file dengan menggunakan algoritma *Breadth First Search* atau *Depth First Search* untuk satu atau seluruh kemunculan file tersebut pada lokasi pencarian. Berikut merupakan algoritma pada FolderCrawler.cs, *Breadth First Search*, dan *Depth First Search*.

4.1.1. Algoritma FolderCrawler.cs

```
procedure FolderCrawler(input inputStartingDirectory, inputFileToFind:  
string, inputAlgorithm: integer, inputfindAll: boolean, inputForm1: Form1)  
{ Program utama folder crawler berdasarkan input pengguna }  
  
KAMUS  
    startingDirectory : string  
    fileToFind : string  
    algorithm : integer  
    findAll : boolean  
    form1 : Form1  
    listOfNode : List<string>  
    parentNode : List<string>  
    childNode : List<string>  
    graph : Graph  
  
{ Graph Nodes Constructor }  
procedure ConstructGraphNodes(input directory: string, input/output  
listOfNode, parentNode, childNode : List<string>)  
  
{ Graph Constructor }  
procedure ConstructGraph(input/output graph: Graph, input parentNode,  
childNode : List<string>)
```

```

{ Proses pencarian }
procedure search()

ALGORITMA
    startingDirectory ← inputStartingDirectory
    fileToFind ← inputFileToFind
    algorithm ← inputAlgorithm
    findAll ← inputFindAll
    form1 ← inputForm1
    listOfNode ← new List<string>()
    parentNode ← new List<string>()
    childNode ← new List<string>()

    ConstructGraphNodes(startingDirectory, listOfNode, parentNode,
    childNode)
    graph ← new Graph("graph")
    ConstructGraph(graph, parentNode, childNode)

    search()

{ Implementasi prosedur Graph Nodes Constructor }
procedure ConstructGraphNodes(input directory: string, input/output
listOfNode, parentNode, childNode : List<string>)

KAMUS LOKAL
    files : array of string
    file : string
    subDirectories : array of string
    subDirectory : string

ALGORITMA
    files ← Directory.GetFiles(directory)
    subDirectories ← Directory.GetDirectories(directory)
    listOfNode.Add(Path.GetFileName(directory))

    foreach (file in files) do
        parentNode.Add(Path.GetFileName(directory))
        childNode.Add(Path.GetFileName(file))
        listOfNode.Add(Path.GetFileName(file))

    foreach (subDirectory in subDirectories) do
        parentNode.Add(Path.GetFileName(directory))
        childNode.Add(Path.GetFileName(subDirectory))
        ConstructGraphNodes(subDirectory, listOfNode, parentNode,
        childNode)

{ Implementasi prosedur Graph Constructor }
procedure ConstructGraph(input/output graph: Graph, input parentNode,
childNode : List<string>)

KAMUS LOKAL
    i: integer

ALGORITMA
    i traversal [0..parentNode.Count]
        graph.AddEdge(parentNode[i], childNode[i])

```

```

{ Implementasi prosedur Search }
procedure search()

KAMUS LOKAL

ALGORITMA
    form1.draw(graph)

    if (algorithm = 1) then
        { BFS }
        BFS bfs = new BFS(form1, this, graph)
        bfs.processBFS()
    else
        { DFS }
        DFS dfs = new DFS(form1, this, graph)
        dfs.processDFS()

```

4.1.2. Algoritma Breadth First Search

```

procedure processBFS()
{ Memulai proses pencarian menggunakan algoritma BFS }

KAMUS
    answerExist      : boolean
    isSolution       : array of boolean
    visited          : array of boolean
    adjchild, adjchild2 : List<integer>
    listOfNode       : List<integer>
    searchPath       : List<integer>
    idxsolution      : List<integer>
    idxqueue         : Queue<integer>
    i, ctidx, ct2idx : integer
    ct               : string
    fc               : FolderCrawler
    form1            : Form1

{ Fungsi untuk mencatat adjacent child dari node }
function returnAdjacentChildNodes(node: string) → List<int>

{ Fungsi untuk mencatat adjacent parent dari node }
function returnAdjacentParentNodes(node: string) → List<int>

{ Fungsi untuk mencari index node childTarget pada listOfNode }
function childIdxInLON(childTarget: string) → integer

{ Prosedur untuk memvisualisasikan langkah-langkah DFS }
procedure visualize()

{ Prosedur untuk mencatat jalur paling pertama ditemukan dari target
menuju root }
procedure trackOnePath(childindex: integer)

{ Prosedur untuk mencatat semua jalur dari target menuju root }

```

```

procedure trackAllPath(childindex: integer)
{Fungsi untuk menentukan apakah pencarian harus dihentikan atau tidak}
function stopCheck(this: DFS) → boolean

ALGORITMA
{ Root dikunjungi }
visited[0] ← true
searchPath.Add(0)

{ Cari child-child dari node yang dikunjungi saat ini }
adjchild ← returnAdjacentChildNodes(listOfNode[0])

{ Lakukan looping pada semua child di adjchild }
i ← 0
while (i < adjchild.Count and !stopCheck()) do
    { Kunjungi child pertama }
    ct ← childNode[adjchild[i]]
    ctidx ← childIdxInLON(ct)
    searchPath.Add(ctidx)

    if (ctidx < this.listOfNode.Count) then
        {Tandai telah dikunjungi, masukkan ke antrian idxqueue}
        idxqueue.Enqueue(ctidx)
        visited[ctidx] ← true

        { Memeriksa apakah node saat ini adalah target }
        if (fc.getFileToFind() = listOfNode[ctidx]) then
            searchPath.Add(ctidx)
            answerExist ← true
            if (stopCheck()) then
                trackOnePath(ctidx)
            else
                this.trackAllPath(ctidx)
                i++;
        else
            i++
    else
        i++

{ Lakukan proses yang sama dengan sebelumnya pada node yang sudah
tercatat pada antrian idxqueue }
while(idxqueue.Count > 0 and !stopCheck()) do
    childTarget ← idxqueue.Dequeue()
    adjchild2 ← returnAdjacentChildNodes(listOfNode[childTarget])
    int j = 0;
    while (j < adjchild2.Count and !stopCheck())){
        ct2 ← childNode[adjchild2[j]]
        ct2idx ← childIdxInLON(ct2)

        { Memeriksa apakah node saat ini sudah

```

```

        pernah dikunjungi }
        if (ct2idx < listOfNode.Count and !visited[ct2idx]) then
            searchPath.Add(ct2idx)
            idxqueue.Enqueue(ct2idx)
            visited[ct2idx] ← true

        { Memeriksa apakah node saat ini adalah target }
        if (fc.getFileToFind() = listOfNode[ct2idx])then
            searchPath.Add(ct2idx)
            answerExist ← true

        { Memeriksa apakah perlu menampilkan
        semua path yang mengarah ke target atau
        tidak }
        if (fc.getfindAll())
            trackAllPath(ct2idx)
        else
            trackOnePath(ct2idx)
            j++
        else
            j++
    else
        j++
{ Menghentikan waktu proses algoritma }
form1.stopwatch.Stop();
form1.writeTimeElapsed();

{ Lakukan visualisasi }
visualize();

```

4.1.3. Algoritma *Depth First Search*

```

procedure processDFS()
{ Memulai proses pencarian menggunakan algoritma DFS }

KAMUS
    answerExist : boolean
    isSolution  : array of boolean
    visited      : array of boolean
    searchPath   : List<int>
    idxsolution : List<int>
    fc          : FolderCrawler
    form1       : Form1

    adjchילד : List<int>
    found      : boolean
    i          : integer
    ctarget    : string
    ctidx      : integer

{ Fungsi untuk mencatat adjacent child dari node }

```

```

function returnAdjacentNodes(node: string) → List<int>

{ Fungsi untuk mencari index node childTarget pada listOfNode }
function childIdxInLON(childTarget: string) → integer

{ Fungsi rekursif untuk algoritma DFS }
function recursiveDFS(nodeIdx: integer) → boolean

{ Prosedur untuk memvisualisasikan langkah-langkah DFS }
procedure visualize()

ALGORITMA
{ root dikunjungi }
visited[0] ← true

{ node dimasukkan ke path }
searchPath.Add(0)
idxsolution.Add(0)

{ mencatat adjacent child dari root }
adjchild ← returnAdjacentNodes(listOfNode[0])

found ← false
i traversal [0..adjchild.Count]
    ctarget ← childNode[adjchild[i]]
    ctidx ← childIdxInLON(ctarget)
    if (ctidx < listOfNode.Count) then
        if (not fc.getfindAll() and answerExist) then
            { do nothing }
        else
            found ← recursiveDFS(ctidx)

{ Menghitung dan mencetak waktu kerja program }
form1.stopwatch.Stop()
form1.writeTimeElapsed()

{ Memvisualisasikan langkah-langkah DFS }
visualize()

{ Implementasi fungsi recursiveDFS }
function recursiveDFS(nodeIdx: integer) → boolean
{ Fungsi rekursif untuk algoritma DFS }

KAMUS
    solutionPath : string
    node         : string
    adjchild     : List<int>
    found        : boolean
    idx          : integer
    i             : integer
    ctarget       : string

```

```

ctidx      : integer

{ Fungsi untuk mencatat adjacent child dari node }
function returnAdjacentNodes(node: string) → List<int>

{ Fungsi untuk mencari index node childTarget pada listOfNode }
function childIdxInLON(childTarget: string) → integer

ALGORITMA
    { node urutan nodeIdx dikunjungi }
    visited[nodeIdx] ← true

    { node dimasukkan ke path }
    searchPath.Add(nodeIdx)
    idxsolution.Add(nodeIdx)

    { pengecekan apakah node ini merupakan file target }
    if (fc.getFileToFind() = listOfNode[nodeIdx]) then
        { node target ditemukan }
        answerExist ← true

        solutionPath ← fc.getStartingDirectory()
        foreach (idx in idxsolution) do
            if (idx ≠ 0 and listOfNode[idx] ≠ fc.getFileToFind())
            then
                node ← listOfNode[idx]
                solutionPath ← solutionPath + "\\" + node
                isSolution[idx] ← true

    { Menambahkan path ke dalam combobox hyperlink }
    form1.addComboBoxElmt(solutionPath)

    if (fc.getfindAll()) then
        idxsolution.Remove(nodeIdx)
        → false
    else
        → true

{ Apabila node ini bukan target, lanjut kunjungi child
  Mencatat adjacent child dari root }
adjchild ← returnAdjacentNodes(listOfNode[nodeIdx])

{ Inisialisasi found dengan false akibat kemungkinan ketidakadaan
  child }
found ← false
i traversal [0..adjchild.Count]
    ctarget ← childNode[adjchild[i]]
    ct2idx ← childIdxInLON(ctarget)
    if (ct2idx < listOfNode.Count) then
        if (not fc.getfindAll() and answerExist) then
            { do nothing }
        else

```

```

        found ← recursiveDFS(ct2idx)

        if (not found) then
            { hapus node dari path pencarian yang valid }
            idxsolution.Remove(nodeIdx)

        → found

```

4.2. Struktur Data

Struktur data pada program ini dibentuk menggunakan sebuah class FolderCrawler, yang mencatat berbagai atribut seperti:

- startingDirectory: menyimpan data folder awal (root)
- fileToFind: menyimpan data target file
- algorithm: mencatat pilihan user mengenai algoritma yang dipilih untuk digunakan
- findAll: mencatat pilihan user mengenai keputusan untuk mencari semua kemunculan yang sesuai atau tidak
- form1: menyimpan data GUI
- listOfNode: menyimpan semua *nodes* dari folder awal
- parentNode: menyimpan semua *nodes* yang berperan sebagai *parent node*
- childNode: menyimpan semua *nodes* yang berperan sebagai *child node*
- graph: menyimpan struktur dari *nodes-nodes* folder dalam bentuk graf
- gviewer: berfungsi untuk menampilkan graf pada GUI

Pada pemrosesan DFS dan BFS, struktur data ini di-pass ke dalam objek yang dibuat dari *class* DFS / BFS untuk melakukan pencarian fileToFind pada listOfNode.

4.3. Tata Cara Penggunaan Program

4.3.1. Cara untuk menjalankan program

4.3.1.1. Melalui Visual Studio

1. Jalankan Visual Studio yang telah terpasang sebelumnya.
2. Buka solution explorer dan pilih file DBFS.sln yang terletak pada folder src/DBFS pada repository ini pada Visual Studio.

3. Klik tombol "Start" pada panel atas dan pastikan DBFS sudah terpilih.
4. Visual Studio akan secara otomatis menjalankan proses build dan menjalankan aplikasi jika build berhasil.

4.3.1.2. Melalui *Executable Code*

1. Pilih dan klik pada file DBFS.exe yang ada pada folder bin pada repository ini.

4.3.2. Cara untuk menggunakan program

1. Jalankan aplikasi DBFS melalui file solution dengan Visual Studio atau melalui executable code.
2. Jika aplikasi berhasil dijalankan akan ditampilkan menu utama dari aplikasi DBFS.
3. Klik tombol "Change folder.." untuk memilih starting directory yang ingin digunakan sebagai direktori awal pencarian file.
4. Masukkan nama file yang ingin dicari beserta extension-nya.
5. Pilih method algorithm yang ingin digunakan dalam proses pencarian file tersebut, yakni menggunakan algoritma Breadth First Search (BFS) atau Depth First Search (DFS).
6. Jika ingin mencari setiap kemunculan file di dalam direktori tersebut, check checkbox "findAll".
7. Untuk memproses dan menampilkan hasil serta visualisasi langkah per langkah, klik tombol "Search". Hasil visualisasi, kemudian, akan tampil pada gviewer.
8. Untuk membuka lokasi ditemukannya file tersebut di dalam direktori, pilih salah satu hyperlink pada dropdown menu dan klik "Open file location". Jika ingin membuka hyperlink di browser, klik "Copy to Clipboard" untuk menyalin path dan tempel pada browser anda.
9. Untuk melihat waktu kerja algoritma, Anda dapat melihat pada textBox time taken (s).

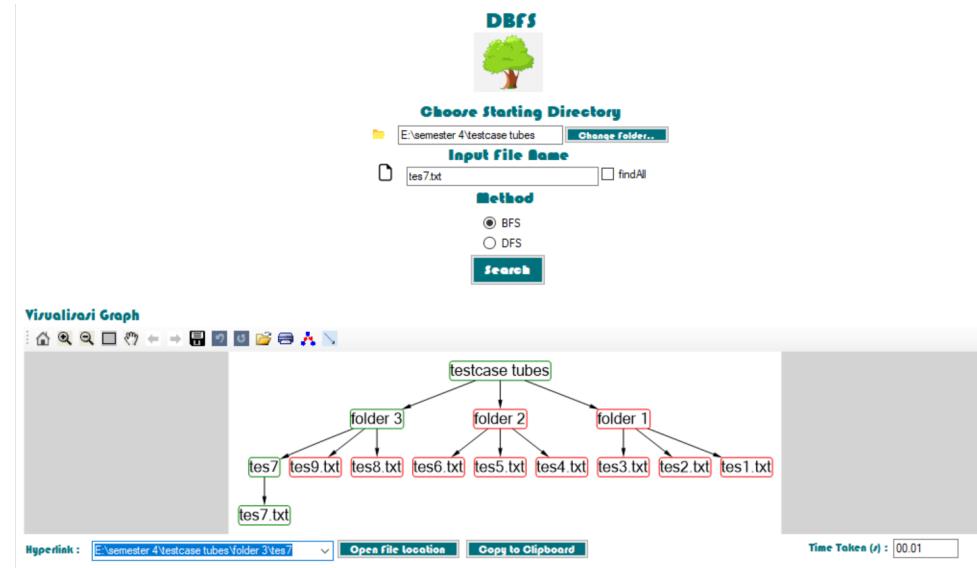
4.4. Hasil Pengujian

4.4.1. Pengujian 1 (BFS)

Pada pengujian pertama diberikan sebuah direktori sebagai berikut.

```
testcase tubes/
├── folder 1/
│   ├── tes1.txt
│   ├── tes2.txt
│   └── tes3.txt
└── folder 2/
    ├── tes4.txt
    ├── tes5.txt
    └── tes6.txt
└── folder 3/
    ├── tes7/
    │   └── tes7.txt
    ├── tes8.txt
    └── tes9.txt
```

Pada pengujian ini, *file* yang dicari memiliki nama “tes7.txt”. Dengan melakukan proses pencarian dengan menggunakan *method algorithm* BFS, hasil yang didapatkan adalah sebagai berikut.



4.4.2. Pengujian 2 (DFS)

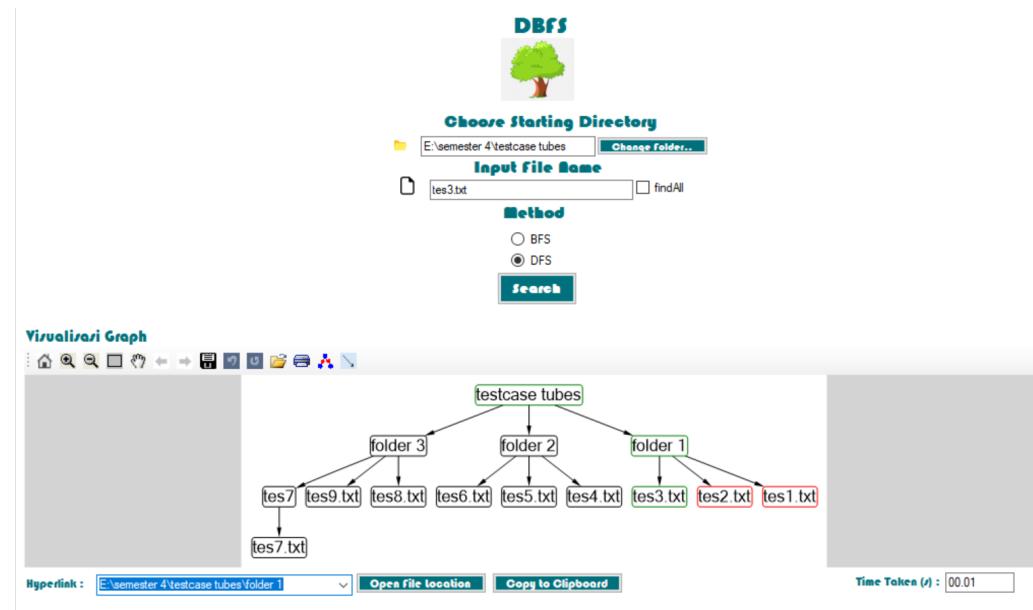
Pada pengujian kedua diberikan sebuah direktori yang sama dengan pengujian pertama, yakni sebagai berikut.

```

testcase tubes/
├─ folder 1/
│  ├─ tes1.txt
│  ├─ tes2.txt
│  └─ tes3.txt
└─ folder 2/
  ├─ tes4.txt
  ├─ tes5.txt
  └─ tes6.txt
└─ folder 3/
  ├─ tes7/
    └─ tes7.txt
  └─ tes8.txt
  └─ tes9.txt

```

Pada pengujian ini, *file* yang dicari memiliki nama “tes3.txt”. Dengan melakukan proses pencarian dengan menggunakan *method algorithm* DFS, hasil yang didapatkan adalah sebagai berikut.



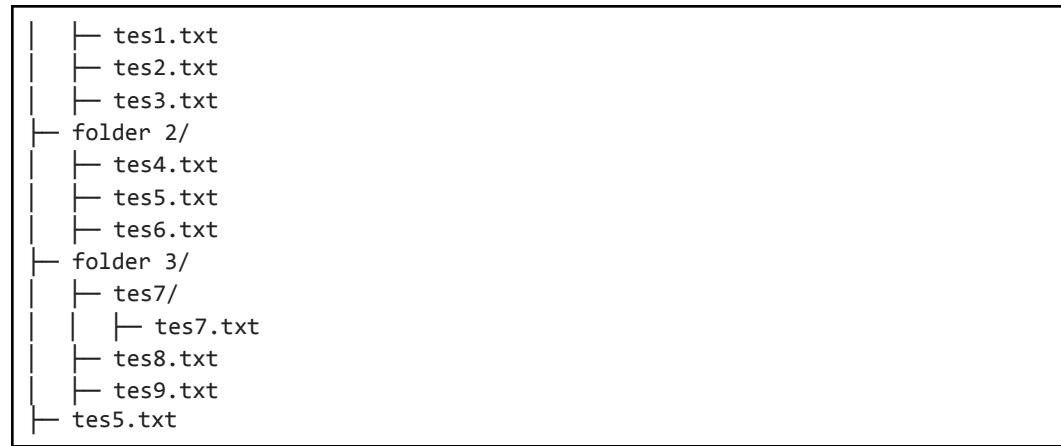
4.4.3. Pengujian 3 (DFS – Find All Occurrence)

Pada pengujian kedua diberikan sebuah direktori yang sama dengan pengujian pertama, namun ditambahkan *file* dengan nama “tes5.txt” pada *folder* “testcase tubes”. Dengan demikian, isi direktori menjadi sebagai berikut.

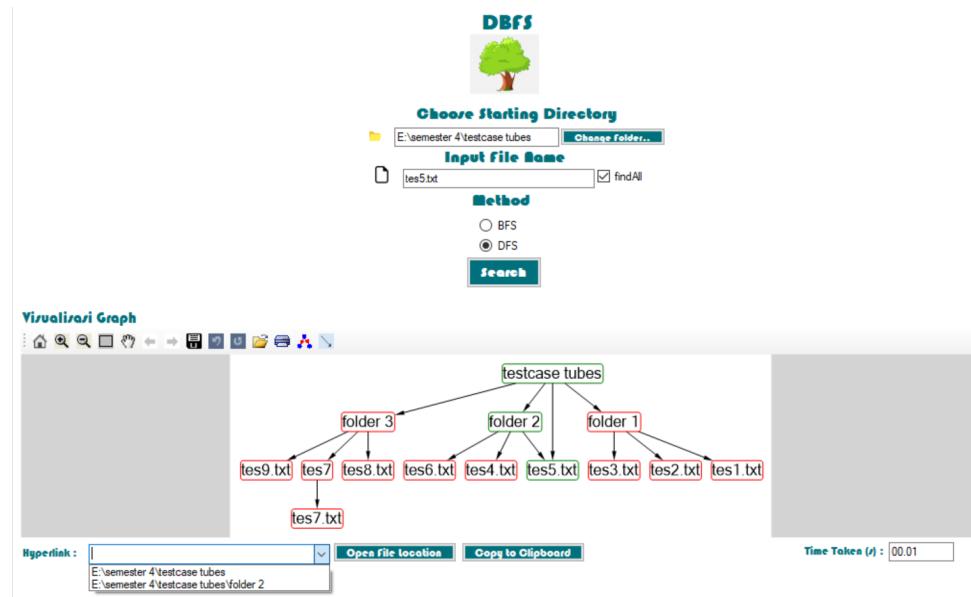
```

testcase tubes/
└─ folder 1/

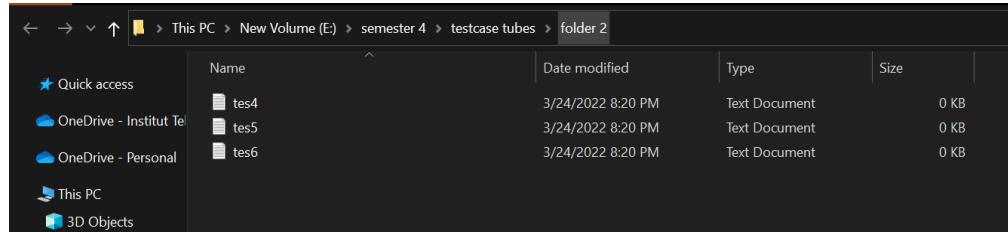
```



Pada pengujian ini, *file* yang dicari memiliki nama “tes5.txt”. Dengan melakukan proses pencarian dengan menggunakan *method algorithm* DFS dan *checkbox* “findAll” di-check, hasil yang didapatkan adalah sebagai berikut.



Pada *dropdown menu/ComboBox hyperlink* ditekan, muncul dua buah pilihan *path* yang menuju *file* yang memiliki nama “tes5.txt”. Ketika sebuah *hyperlink* dipilih, misalnya “E:\semester 4\testcase tubes\folder 2” dan tombol “Open file location” diklik, program akan membuka folder pada path tersebut dengan tampilan sebagai berikut.



Selain menekan tombol “Open file location”, pengguna juga dapat menekan tombol “Copy to Clipboard” untuk menyalin path ke dalam *clipboard* untuk kemudian ditempel pada kolom pencarian di *browser* masing-masing. Berikut adalah tampilan jika membuka *path* di dalam *browser*.

The screenshot shows a browser window with the URL `E:\semester 4\testcase tubes\folder 2\` in the address bar. The page content is a table with the following data:

Nama	Ukuran	Tanggal Dimodifikasi
tes4.txt	0 B	24/03/22 20.20.21
tes5.txt	0 B	24/03/22 20.20.26
tes6.txt	0 B	24/03/22 20.20.32

4.5. Analisis Desain Solusi Algoritma BFS dan DFS berdasarkan Hasil Pengujian

Desain solusi Algoritma BFS dan DFS yang diterapkan memiliki kelebihan dan kekurangan masing-masing antara algoritma BFS dan DFS. Algoritma DFS perlu dilakukan secara rekursif sehingga sangat memudahkan *path tracking* karena perjalanan yang dilakukan oleh program dapat dicatat setiap fungsi rekursifnya dieksekusi. Implementasi algoritma DFS pada program ini jauh lebih sulit ketika tidak memahami struktur yang menyimpan data *nodes*-nya.

Di samping itu, algoritma BFS hanya perlu melakukan iterasi berdasarkan urutan antrian yang dikunjungi, sehingga pencarian hanya membutuhkan *looping* yang mengiterasi *node* pada antrian. Masalah utama pada BFS adalah pencatatan *path* dari *root* ke target karena secara perilaku, pencarian dengan algoritma BFS tidak merepresentasikan rantai jalan dari *starting point* ke *target*, sehingga solusi yang program ini gunakan adalah dilakukannya *backtracking* dari *node target* ke *root*.

Pada program ini, struktur pada program ini tidak dibuat menyerupai graf, tetapi list yang bisa merepresentasikan graf sehingga akan sangat sulit digunakan apabila dilihat dengan algoritmanya secara langsung, sehingga dibuat beberapa fungsi yang

memberikan nuansa graf pada struktur data tersebut, seperti fungsi untuk mencari child dari sebuah node.

Di samping itu, terdapat sebuah permasalahan pada program kami, yakni ketika memperoleh masukan direktori yang memiliki *folder* dengan nama ganda sehingga menyebabkan program menganggap bahwa kedua *folder* merupakan *node* atau *folder* yang sama sehingga dapat menyebabkan permasalahan dalam proses pencarian BFS dan DFS.

BAB V

KESIMPULAN DAN SARAN

5.1. Kesimpulan

Algoritma DFS dan BFS merupakan suatu algoritma yang mencari node pada suatu graf, dimulai dari node *root* hingga ke node *destination*. Dengan memanfaatkan kedua algoritma tersebut kelompok kami berhasil untuk aplikasi sederhana yang digunakan untuk mencari letak suatu file.

5.2. Saran

Saran yang dapat kami berikan untuk pengembangan aplikasi ini adalah dengan mempercantik dan menambah kreativitas dalam desain GUI aplikasi ini. Dengan mempercantik dan menambah kreativitas tersebut diharapkan *user experience* dari pengguna juga akan meningkat. Saran lainnya adalah akan lebih baik untuk mengerjakan penulisan struktur data dengan algoritma secara bersamaan karena besar kemungkinan ada kesalahpahaman antara dua orang yang mengerjakan hal tersebut. Selain itu kami juga memberikan saran terkait tugas besar ini, kami berharap kedepannya tugas seperti ini tetap dipertahankan karena tugas ini menuntut mahasiswa untuk bereksplorasi terkait dengan bidang studinya.

DAFTAR PUSTAKA

Munir, Rinaldi. (2021). Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1) (Versi baru 2021). Institut Teknologi Bandung.<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf> . Diakses pada 14 Maret 2022.

Munir, Rinaldi. (2021). Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 2) (Versi baru 2021). Institut Teknologi Bandung. <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf> . Diakses pada 14 Maret 2022.

LAMPIRAN

Link Github:

https://github.com/ilhampratama2109/Tubes2_13520041

Link Youtube:

<https://youtu.be/y6KtIkVdmvM>