

**Laporan Tugas Besar 2 IF2211 Strategi Algoritma**  
**Semester II Tahun Akademik 2023/2024**

**WikiRace Solver Menggunakan Algoritma IDS dan BFS**



**Disusun Oleh:**

Eduardus Alvito Kristiadi 13522004

Bryan Cornelius Lawrence 13522033

Vanson Kurnialim 13522049

**Sekolah Teknik Elektro dan Informatika**

**Institut Teknologi Bandung**

**2024**

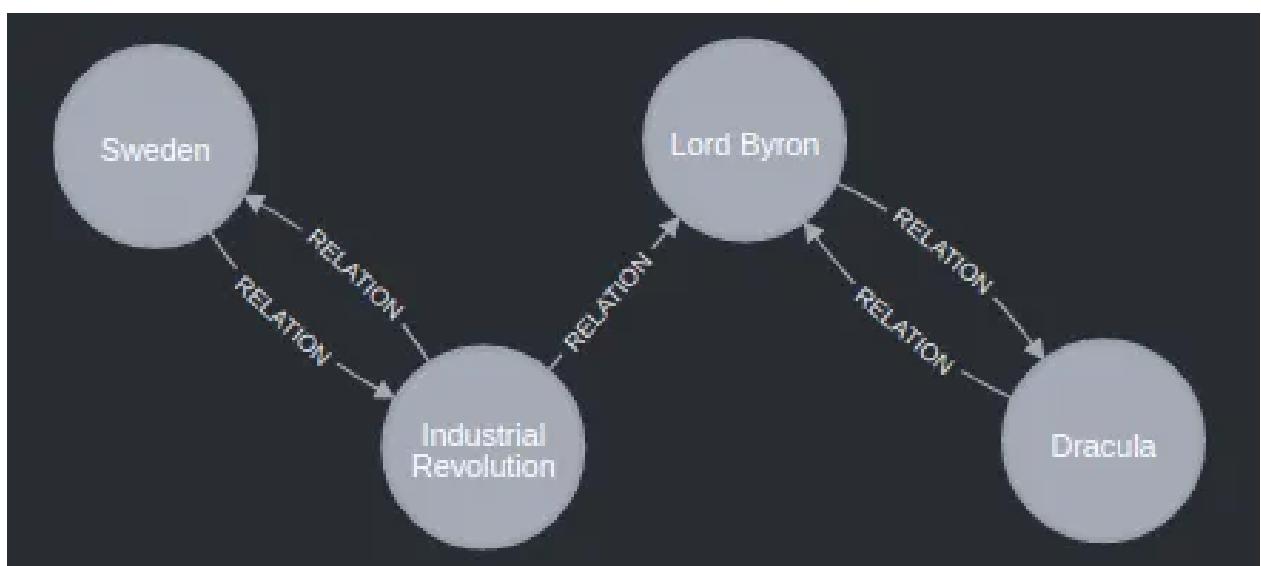
# **DAFTAR ISI**

<b>DAFTAR ISI.....</b>	<b>2</b>
<b>BAB I</b>	
<b>DESKRIPSI TUGAS.....</b>	<b>3</b>
<b>BAB II</b>	
<b>LANDASAN TEORI.....</b>	<b>4</b>
A. Dasar Teori.....	4
B. Aplikasi Web.....	4
<b>BAB III</b>	
<b>ANALISIS PEMECAHAN MASALAH.....</b>	<b>6</b>
A. Langkah-Langkah Pemecahan Masalah.....	6
B. Proses Pemetaan Masalah.....	6
C. Fitur Fungsional dan Arsitektur Aplikasi Web.....	7
D. Ilustrasi Kasus.....	8
<b>BAB IV</b>	
<b>IMPLEMENTASI DAN PENGUJIAN.....</b>	<b>10</b>
A. Spesifikasi Teknis Program.....	10
B. Tata Cara Penggunaan Program.....	17
C. Hasil Pengujian.....	17
D. Analisis Hasil Pengujian.....	20
<b>BAB V</b>	
<b>KESIMPULAN DAN SARAN.....</b>	<b>21</b>
<b>LAMPIRAN.....</b>	<b>22</b>
GitHub.....	22
<b>DAFTAR PUSTAKA.....</b>	<b>23</b>

# BAB I

## DESKRIPSI TUGAS

WikiRace atau Wiki Game adalah permainan yang melibatkan Wikipedia, sebuah ensiklopedia daring gratis yang dikelola oleh berbagai relawan di dunia, dimana pemain mulai pada suatu artikel Wikipedia dan harus menelusuri artikel-artikel lain pada Wikipedia (dengan mengeklik tautan di dalam setiap artikel) untuk menuju suatu artikel lain yang telah ditentukan sebelumnya dalam waktu paling singkat atau klik (artikel) paling sedikit.



Gambar 1. Ilustrasi Graf WikiRace

(Sumber: [https://miro.medium.com/v2/resize:fit:1400/1\\*jxmEbVn2FFWybZsIicJCWQ.png](https://miro.medium.com/v2/resize:fit:1400/1*jxmEbVn2FFWybZsIicJCWQ.png))

WikiRace Solver yang akan dibuat akan menghasilkan hasil rute pencarian terdekat menggunakan algoritma yang dapat ditentukan oleh pengguna. Tampilan WikiRace Solver akan berbentuk web yang akan menerima awal dan tujuan judul penjelajahan serta tipe algoritma yang ingin digunakan.

## BAB II

# LANDASAN TEORI

### A. Dasar Teori

Pada sebuah graf, terdapat dua pendekatan yang sistematis untuk mengunjungi simpul-simpul di dalam graf. Pendekatan pertama adalah pencarian melebar (*breadth first search/BFS*). BFS merupakan pendekatan dengan mengunjungi suatu simpul kemudian mencari seluruh simpul lainnya yang bertetangga dengan simpul tersebut untuk dilakukan proses yang sama sampai seluruh simpul dikunjungi. Pendekatan ini menggunakan struktur data *queue* karena simpul yang lebih dulu ditemukan akan dikunjungi terlebih dahulu. Pendekatan kedua adalah pencarian mendalam (*depth first search/DFS*) adalah pendekatan dengan mengunjungi simpul tetangga pertama yang ditemukan dan terus dilakukan sampai sampai simpul habis, kemudian mundur (*backtracking*) ke simpul hidup sebelumnya untuk melakukan proses yang sama. Pendekatan ini menggunakan struktur data *stack* karena simpul yang ditemukan terakhir akan dikunjungi terlebih dahulu.

Pendekatan DFS memiliki beberapa pendekatannya sendiri yaitu pencarian mendalam terbatas (*depth limited search/DLS*) dan pencarian berulang mendalam (*iterative deepening search/IDS*). DLS merupakan DFS yang dibatasi dengan kedalaman tertentu saja sehingga pencarian akan dihentikan jika telah mencapai batas kedalaman tersebut. IDS merupakan variasi DFS yang mengulang secara iteratif dari *depth 0* sampai *depth goal node*. Pencarian akan dilakukan dari *depth 0*, jika tidak ditemukan *goal node*, maka akan diulang lagi pencarian dari awal namun dengan batas *depth 1*. Proses akan diulang sedemikian sampai ditemukan *goal node*-nya. Pada WikiRace Solver ini, algoritma DFS tidak mungkin atau tidak optimal digunakan karena *depth* atau pembangkitan anak dari *node* adalah *infinite* atau tidak terbatas. Sehingga diperlukan suatu batasan untuk menghentikan proses pencarian. Maka dari itu, IDS merupakan variasi algoritma DFS yang tepat untuk diterapkan.

### B. Aplikasi Web

Dalam pembuatan sebuah *website*, terbagi menjadi 2 bagian yang sama-sama penting yaitu *frontend* dan *backend*. *Frontend* mengurus segala hal yang akan berhubungan dengan pengguna. Mulai dari tombol-tombol ataupun kolom untuk menerima *input* pengguna serta fitur interaktif lainnya. Dikarenakan berhubungan langsung dengan pengguna, tampilan *website* perlu diperhatikan agar lebih menarik. Secara umum, dalam sebuah *website* digunakan HTML sebagai pembangun struktur halaman web, yang mencakup membuat header, footer, navigasi, hingga kontennya, CSS yang berfungsi untuk memperindah laman

*website* agar lebih menarik bagi pengguna, dan juga bahasa pemrograman seperti Java Script untuk meng-*handle* fitur-fitur yang lebih kompleks dan agar *website* lebih interaktif.

*Backend* mengurus proses-proses yang dikerjakan oleh web yang tidak terlihat oleh pengguna. *Backend* merupakan program yang menjalankan program yang nantinya ditampilkan oleh *frontend* untuk pengguna. Tidak seperti *frontend* yang mengurus tampilan dan *event-event* yang dilakukan oleh pengguna, *backend* menjadi otak yang melakukan perhitungan dan logika-logika yang rumit.

*Frontend* dan *backend* dihubungkan oleh API. Melalui API, *frontend* dapat mengirimkan permintaan (*request*) kepada *backend* untuk diproses. Kemudian, *backend* akan mengirimkan tanggapan (*response*) untuk ditampilkan oleh *frontend*. Metode yang digunakan terdapat pada protokol HTTP, yaitu metode *GET* dan metode *POST*. *GET* dilakukan untuk mengambil sumberdaya dari server dan *POST* dilakukan untuk mengirim sumberdaya ke server.

Meskipun *frontend* dan *backend* memiliki fungsi yang sangat berbeda. Sebuah *website* memerlukan keduanya untuk dapat menghasilkan sebuah *website* yang berfungsi dengan baik. Website tanpa *frontend* tidak akan memiliki tampilan yang menarik bahkan dapat membuat pengguna kesulitan menggunakanannya. Sedangkan *website* tanpa *backend* tidak dapat mengerjakan tugas-tugas yang kompleks.

## **BAB III**

# **ANALISIS PEMECAHAN MASALAH**

### **A. Langkah-Langkah Pemecahan Masalah**

Langkah pemecahan masalah *WikiRace Solver* sebagai berikut.

1. Kunjungi pranala awal dan mengambil setiap elemen a[href] yang mengarah ke laman wikipedia lainnya.
2. Buang laman Wikipedia umum yang didapatkan seperti laman *Category*, *Help*, *Wikipedia*, dan *Main Page*.
3. Lakukan pengecekan judul artikel dari pranala karena pranala dapat memiliki judul artikel yang berbeda.
4. Cek pranala dan juga judul artikel. Ketika ditemukan pranala yang menjadi tujuan, hentikan proses pencarian karena solusi telah didapatkan.
5. Proses pencarian akan dilakukan pada pranala hasil pencarian terus menerus sampai ditemukan artikel tujuan. Pemilihan pranala yang dicari sesuai dengan algoritma yang digunakan.

### **B. Proses Pemetaan Masalah**

Pada *WikiRace Solver*, model yang digunakan untuk mencari adalah BFS dan DFS dengan pembentukan pohon dinamis melalui pembangkitan status pada setiap prosesnya sebagai berikut:

1. Elemen untuk algoritma IDS
  - a. *Node* awal, judul artikel yang *diinput* oleh pengguna.
  - b. *Node* tujuan, judul artikel yang *diinput* oleh pengguna.
  - c. *Node* yang merepresentasikan judul laman Wikipedia
  - d. *List* yang berisi hasil pembangkitan anak suatu *Node*.
  - e. *Matrix* yang berisi jalur pencarian atau rute dari *Node* awal menuju *Node* tujuan.
  - f. *Depth* yang merupakan pembatas pencarian, akan berubah seiring berjalan proses pencarian.
  - g. *boolean* variabel untuk menentukan mode pencarian *single path* atau *multiple path*. (digunakan untuk pengajaran bonus)

2. Elemen untuk algoritma BFS
  - a. *Node* yang berisi pranala dan kedalaman pranala tersebut.
  - b. *Node* awal, judul artikel permulaan yang di-*input* oleh pengguna.
  - c. *Node* tujuan, judul artikel akhir yang di-*input* oleh pengguna.
  - d. Sebuah *queue* yang berisi *node-node* yang belum dikunjungi dan akan dikunjungi.
  - e. Sebuah *map* yang memetakan pohon pembangkitan. *Key* dari *map* adalah pranala sebagai simpul anak dan *value* dari *map* adalah pranala yang menjadi *parent*. *Root* ditandai dengan value “start”.
  - f. *Matrix* yang berisi jalur pencarian dari *node* awal ke *node* tujuan.
  - g. Variabel *integer* *max\_depth* yang menjadi batasan kedalaman ketika ditemukan jalur terdekat.
  - h. Variabel *boolean* *single\_path* yang menandakan pencarian satu jalur saja atau lebih.

## C. Fitur Fungsional dan Arsitektur Aplikasi Web

Website memiliki berbagai fitur fungsional seperti berikut :

1. Kolom pengisian judul awal
2. Kolom pengisian judul tujuan
3. *Auto suggestion* dalam pengisian kolom judul awal dan judul tujuan
4. Tombol memilih algoritma BFS atau IDS
5. Tombol memilih hasil *single path* atau *multiple path*
6. Tombol untuk mulai menjalankan program

Website dibuat menggunakan React.JS dengan *framework* styling tailwind CSS. Halaman utama website merupakan halaman pertama yang muncul ketika masuk ke laman website dan berisikan fungsi utama program. Pada laman ini terdapat semua fitur fungsional yang telah disebutkan di atas. Pada kanan atas *main page* ini juga terdapat beberapa *page* lain yang dapat dikunjungi seperti penjelasan konsep BFS dan IDS, *How to use*, dan laman *About us*.

## D. Ilustrasi Kasus

Diberikan judul awal “Nasi padang” dan judul tujuan “Cooked rice”. Maka proses akan melakukan *scraping* pada judul awal. Jika algoritma yang dipilih merupakan BFS, hasil *scraping* akan dimasukkan ke dalam sebuah *queue* yang akan diproses lebih lanjut sesuai urutan *queue*. Untuk IDS, hasil akan disimpan ke dalam sebuah *list* dan selanjutnya akan memanggil fungsi rekursif untuk men-*scraping* semua hasil dalam *list* secara terpisah.

Pengecekan hasil dilakukan dengan *string compare* antara judul tujuan dengan hasil *scraping*. Karena hasil *scraping* merupakan elemen HTML dari a[href], pranala tersebut bisa jadi merupakan pranala *redirect* yang memiliki judul artikel yang berbeda. Sehingga pengecekan kedua akan dilakukan saat proses mengunjungi pranala tersebut dan men-*compare* kembali judul tujuan dengan judul artikel agar didapatkan hasil yang lebih akurat. Misal salah satu pranala hasil *scraping* “Nasi padang” adalah “Steamed\_rice”, namun pranala ini merupakan *redirect* menuju judul tujuan “Cooked rice”. Maka pengecekan pertama tidak akan berhasil. Ketika pengecekan kedua dilakukan, barulah didapatkan bahwa di *depth* 1 ini, rute hasil telah ditemui. Implementasi pengecekan hasil seperti ini digunakan oleh kedua jenis algoritma.

Terdapat pula penggunaan konkurensi *go routine* pada BFS maupun IDS, Pada BFS, konkurensi dilakukan pada pemrosesan queue yang dibatasi sesuai pengaturan, misal 200. Maka, pada satu waktu proses hanya bisa melakukan 200 *scraping*. Untuk IDS, konkurensi dilakukan pada saat hasil *scraping* ingin diproses menuju fungsi rekursif iterasi berikutnya. Terdapat pembatasan misal 5 buah konkurensi pada setiap aktivasi sehingga proses dapat ditahan agar tidak memakan memori yang terlalu banyak pada sekali waktu. Pada kasus ini, jika menggunakan IDS, maka jika hasil *scraping* “Nasi padang” berjumlah 10, hanya 5 yang akan di konkurenksikan untuk berproses secara independen.

Pemberhentian proses dilakukan secara berbeda bergantung pada BFS dan IDS beserta dengan jenis hasil yang diinginkan, *single path* atau *multiple path*. Untuk algoritma BFS *single path*, proses akan berhenti ketika hasil rute telah didapatkan dengan langsung me-*break* proses. IDS *single path* juga me-*break* seluruh *go routine* yang ada ketika telah ditemukan rute hasil. Pada BFS *multiple path*, proses akan membatasi *depth* maksimal menjadi *depth* dari hasil yang pertama kali ditemui, sehingga proses akan mendapatkan seluruh rute terpendek. Untuk IDS *multiple path*, proses tinggal memeriksa apakah telah ditemukan rute hasil pada setiap akhir iterasi karena pada dasarnya IDS memang terbagi-bagi sesuai dengan iterasi. Selain kondisi pemberhentian yang telah dijelaskan di atas, seluruh

proses BFS maupun IDS akan berhenti ketika waktu telah mendekati 5 menit agar program tidak berjalan terlalu lama dan juga sesuai dengan spesifikasi permasalahan.

Diterapkan pula optimasi *cache* dengan menggunakan *library* dari *go colly*. *Caching* ini berguna untuk mengurangi waktu eksekusi saat pranala yang sudah pernah *divisit* ingin *divisit* kembali. Jika pada percobaan pertama “Nasi padang” menuju “Cooked rice” belum tersimpan dalam *cache*, maka percobaan selanjutnya pasti akan jauh lebih cepat karena data kunjungan telah tersimpan dalam *cache*. Tentunya *cache* yang digunakan tidaklah *infinite* melainkan terbatas dengan pranala paling terakhir digunakan saja yang tersimpan.

# BAB IV

## IMPLEMENTASI DAN PENGUJIAN

### A. Spesifikasi Teknis Program

Program *backend* kami dibagi menjadi tiga bagian, yaitu BFS, IDS, dan utility. Berikut merupakan daftar *pseudocode* fungsi dan prosedur di dalam utility yang digunakan oleh kedua algoritma :

```
function isIn(lis: array of string, s: string) -> boolean

KAMUS LOKAL
ALGORITMA
    for (tiap string dalam lis) do
        if (ditemukan elemen lis yang sama dengan s) then
            return true
        endif
    endfor
    return false
```

```
function cutLink(l: string, i: integer) -> string

KAMUS LOKAL
ALGORITMA
    return l[:i]
```

Fungsi *isIn* memberikan *true* jika terdapat sebuah *string* di dalam suatu *array of string*. Fungsi *cutLink* berfungsi untuk memotong *array of string* sesuai dengan parameternya.

Berikut merupakan daftar *pseudocode* utility untuk kepentingan algoritma BFS :

```
type Node

type Node : < link : string {sebagai pranala},
            depth : integer {sebagai kedalaman} >
```

```
function isIn(lis: array of Node, s: string) -> boolean
```

**KAMUS LOKAL****ALGORITMA**

```
for (tiap Node dalam lis) do
    if (ditemukan link dari Node yang sama dengan s) then
        return true
    endif
endfor
return false
```

```
function isInPath(lis: array of array of string, s: array of string) ->
boolean
```

**KAMUS LOKAL****ALGORITMA**

```
for (tiap array of string dalam lis) do
    if (ditemukan array of string yang sama dengan s) then
        return true
    endif
endfor
return false
```

```
function makePath(m: map[string]string, now, des: string) -> array of
string
```

**KAMUS LOKAL**

```
ret: array of string
parent: string
```

**ALGORITMA**

```
(masukkan now ke dalam ret)
while parent != "start" do
    parent = m[now]
    (masukkan parent ke dalam ret)
    now = parent
endwhile
(reverse ret)
(masukkan des ke dalam path)
return ret
```

```

procedure validasiLinkBFS(input/output q: array of Node, m:
map[string]string)

```

---

**KAMUS LOKAL**

```

flag: boolean
title: string
parent: string
l: string
current: integer
n: integer
tempPath: array of string
tempNode: Node

```

**ALGORITMA**

```

total_link_visited = total_link_visited + 1
flag = false
l = q[0].link
current = q[0].depth
(kunjungi link l)
if (judul artikel yang dikunjungi = destination) then
    flag = true
    max_depth = current-1
    tempPath = makePath(m, m[l], judul_artikel)
    (masukkan tempPath ke dalam path_found)
endif

if !flag and (menemukan a[href]="/wiki") then
    if not isInNode(q, link_ditemukan) && (link_ditemukan tidak ada
        di map)then
        if(link_ditemukan = destination) then
            max_depth = current
            tempPath = makePath(m, l, link_ditemukan)
            if not isInPath(path_found,tempPath) then
                (masukkan tempPath ke dalam path_found)
            endif
        else
            m[link_ditemukan] = 1
            tempNode.link = l; tempNode.depth = current
            (Masukkan tempNode ke dalam q)
        endif
    endif
endif

```

```
endif
```

*Node* didefinisikan sebagai tipe struktur data dengan 2 atribut, *Link* yang bertipe *string* dan *Depth* yang bertipe *integer*. *Node* ini berguna untuk mengetahui status dari *Link* atau suatu pranala. *isIn* disini mengembalikan *true* jika sebuah *string* berada di dalam suatu *array of Node*. Fungsi *isInPath* juga sama, namun yang diperiksa adalah *Matrix of string*. Fungsi *makePath* membuat sebuah *list of string* yang merupakan jalur dari sebuah *Node* menuju *parentnya* sampai dengan *Node* awal atau judul awal. Prosedur *validasiLinkBFS* merupakan komponen penting dari algoritma ini, karena pemeriksaan apakah hasil *scrape* suatu pranala merupakan *Node* tujuan akan divalidasi disini. Jika iya, maka jalur dari judul awal sampai tujuan akan disimpan.

Berikut merupakan daftar *pseudocode* utility untuk kepentingan algoritma IDS:

```
procedure scraping(input/output s: array of string)  
  
KAMUS LOKAL  
n: integer  
  
ALGORITMA  
if (menemukan a[href]="/wiki") then  
    if not isIn(s, link_ditemukan) then  
        (masukkan link ke dalam s)  
    endif  
endif
```

```
procedure validasiLinkIDS(input/output l: string, s: array of string,  
input new_path_of_url: array of string)  
  
KAMUS LOKAL  
flag: boolean  
title: string  
  
ALGORITMA  
total_link_visited = total_link_visited + 1  
flag = false  
if (judul artikel yang dikunjungi = destination) then  
    flag = true  
    new_path_of_url[len(new_path_of_url)-1] = judul_artikel  
    (masukkan new_path_of_url ke dalam path_found)  
if not flag then
```

```

scraping(s)
endif

```

Procedure validasiLinkIDS memeriksa apakah suatu pranala merupakan judul tujuan atau bukan. Jika iya maka akan langsung disimpan dalam *list* rute hasil. Jika tidak, maka akan dipanggil procedure scraping terlebih dahulu yang memeriksa apakah hasil *scrape* pranala tersebut merupakan tujuan. Jadi dilakukan 2 kali target pengecekan yang berbeda.

Berikut merupakan *pseudocode* untuk algoritma BFS:

```

procedure BFS()

KAMUS LOKAL
    unvisitedQueue: array of Node
    startNode: Node
    max_depth: integer
    visitedMap: map[string]string

ALGORITMA
    total_link_visited = 0
    max_depth = 100
    startNode.link = link_awal
    startNode.depth = 0
    (masukkan startNode ke dalam unvisitedQueue)
    visitedMap[link_awal] = "start"
    while (waktu < 4,5 menit) and len(unvisitedQueue) > 0 and
        unvisitedQueue[0].depth < max_depth do
            validasiLinkBFS(unvisitedQueue,visitedMap)
            if single_path and len(path_found) > 0 then
                break
        endwhile

```

Prosedur utama dalam algoritma BFS ini akan mengatur seluruh proses pengecekan, mulai dari giliran pengecekan, berapa banyak yang dicek, kondisi pengecekan, dan kapan pengecekan selesai.

Berikut merupakan *pseudocode* untuk algoritma IDS:

```

procedure dls(input/output path_of_url: array of string, input
url_scraped: string, iterasi: integer)

```

**KAMUS LOKAL**

```
list_of_url: array of string
new_path_of_url: array of string
j: integer

ALGORITMA
    (masukkan path_of_url ke dalam new_path_of_url)
    validasiLinkIDS(url_scraped, list_of_url, new_path_of_url)

    iterasi = iterasi - 1
    for j <- 0 to len(list_of_url) do
        if (single_path and len(path_found) > 0) or (waktu < 4,5 menit)
            then
                break
            endif
        if list_of_url[j] = destination then
            (masukkan paht_of_url ke dalam new_path_of_url)
            (masukkan list_of_url[j] ke dalam new_path_of_url)
            (masukkan new_paht_of_url ke dalam path_found)
        else
            if iterasi > 0
                (masukkan paht_of_url ke dalam new_path_of_url)
                (masukkan list_of_url[j] ke dalam new_path_of_url)
                dls(new_path_of_url, list_of_url[j], iterasi)
            endif
        endif
    endfor
```

**procedure** central()**KAMUS LOKAL**

```
path_of_url: array of string
new_path_found: array of array of string
iterasi: integer
depth: integer
```

**ALGORITMA**

```
if start = destination then
    (masukkan [start] ke dalam path_found)
else
    (masukkan start ke dalam path_of_url)
    while true do
```

```

if (len(path_found) > 0) then
    if not single_path and len(path_found) > 0 then
        depth = 100
        for i<-0 to len(path_found) do
            if len(path_found[i]) < depth then
                depth = len(path_found[i])
            endif
        endfor
        for j<-0 to len(path_found) do
            if len(path_found[j]) = depth then
                (masukkan path_found[j] ke new_path_found)
            endif
        endfor
        path_found = new_path_found
    endif
    break
endif
if iterasi = 5 then break
endif
total_link_visited = 0
iterasi = iterasi + 1
dls(path_of_url,start,iterasi)
endwhile
endif

```

**procedure** IDS()

**KAMUS LOKAL**

**ALGORITMA**

central()

Prosedur IDS hanya digunakan sebagai inisialisasi saja pada praktiknya dan langsung memanggil prosedur central. Dalam prosedur ini, central akan mengatur kapan proses selesai dan secara inkrement menambah batasan *depth* untuk memanggil prosedur dls. Prosedur dls ini merupakan *engine* dari algoritma IDS ini dan bersifat rekursif. Di dalamnya, ia akan mengatur percabangan *scraping* dan juga pemanggilan prosedur-prosedur pemeriksaan. Ketika hasil yang sesuai sudah didapatkan, maka dls akan berhenti atau diberhentikan oleh central.

Struktur data yang digunakan dalam program ini adalah *List*, *Queue*, *Matrix*, dan *Map* untuk menyimpan banyak data. Ada juga ADT *Node* yang dibuat untuk memeriksa kedalaman pencarian dan digunakan di dalam *Queue* pada algoritma BFS. Selain yang telah disebutkan, struktur data yang digunakan adalah primitif atau bawaan dari *Go* seperti *string*, *int*, dan lainnya.

## B. Tata Cara Penggunaan Program

Berikut merupakan tata cara menjalankan program:

1. Pastikan Requirement di atas sudah terinstall dengan benar (Golang, Gocolly, npm, dan React).
2. Clone repository ini dengan command “git clone [https://github.com/Edvardesu/Tubes2\\_Dominus-Vobiscum.git](https://github.com/Edvardesu/Tubes2_Dominus-Vobiscum.git)”
3. Masuk ke folder website dengan perintah “cd website”
4. Jalankan website dengan command “npm run dev”
5. Masuk ke folder backend dengan command “cd src/backend”
6. Jalankan command berikut “go run scrap.go BFS.go IDS.go”
7. Buka website pada localhost
8. Masukkan artikel awal pada input box di atas dan artikel tujuan pada input box bawah
9. Pilih algoritma yang ingin dijalankan (BFS atau IDS)
10. Jika ingin mencari hanya 1 path, centang kotak “Single Path”
11. Tekan tombol “Sikat!!!”
12. Tunggu hingga hasil ditampilkan

## C. Hasil Pengujian

### 1. Kasus Kedalaman 0

Input	Hasil
	<b>BFS</b> 
	<b>IDS</b> 

### 2. Kasus kedalaman 1

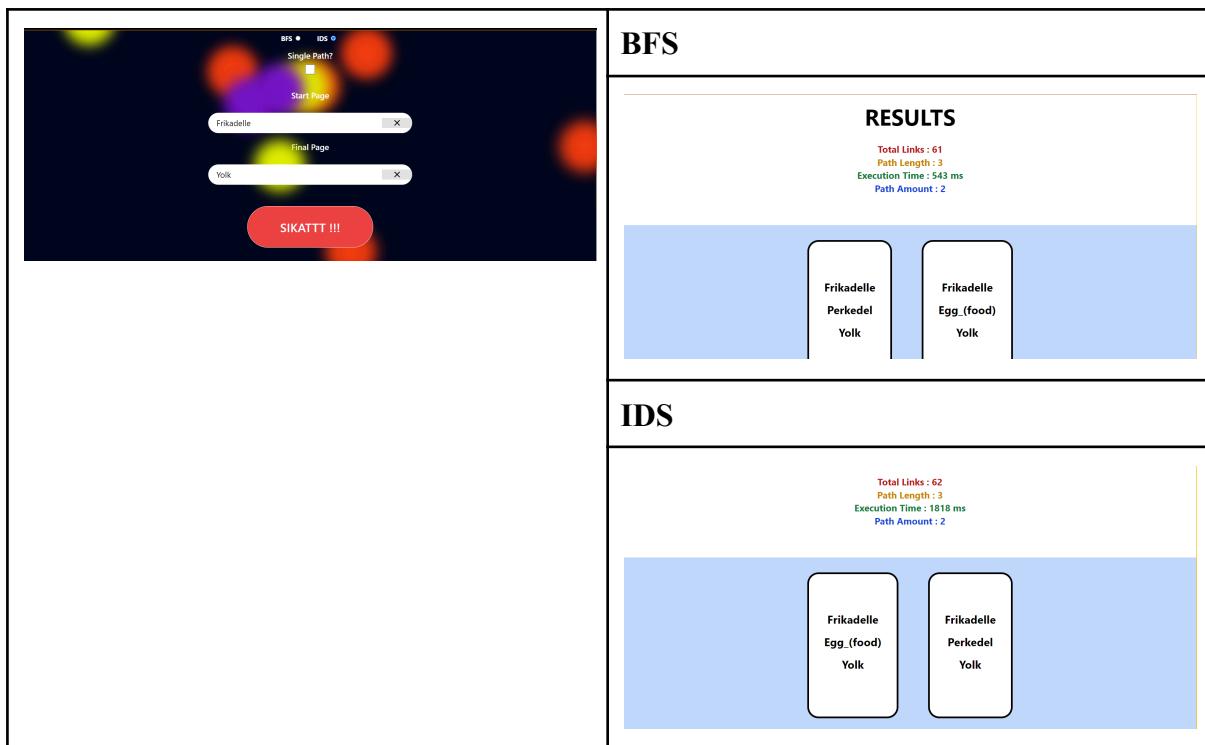
Input	Hasil
	<p><b>BFS</b></p> <p><b>RESULTS</b></p> <ul style="list-style-type: none"> <li>Total Links : 1</li> <li>Path Length : 2</li> <li>Execution Time : 0 ms</li> <li>Path Amount : 1</li> </ul>
	<p><b>IDS</b></p> <p><b>RESULTS</b></p> <ul style="list-style-type: none"> <li>Total Links : 1</li> <li>Path Length : 2</li> <li>Execution Time : 230 ms</li> <li>Path Amount : 1</li> </ul>

### 3. Kasus kedalaman 1 (*redirect*)

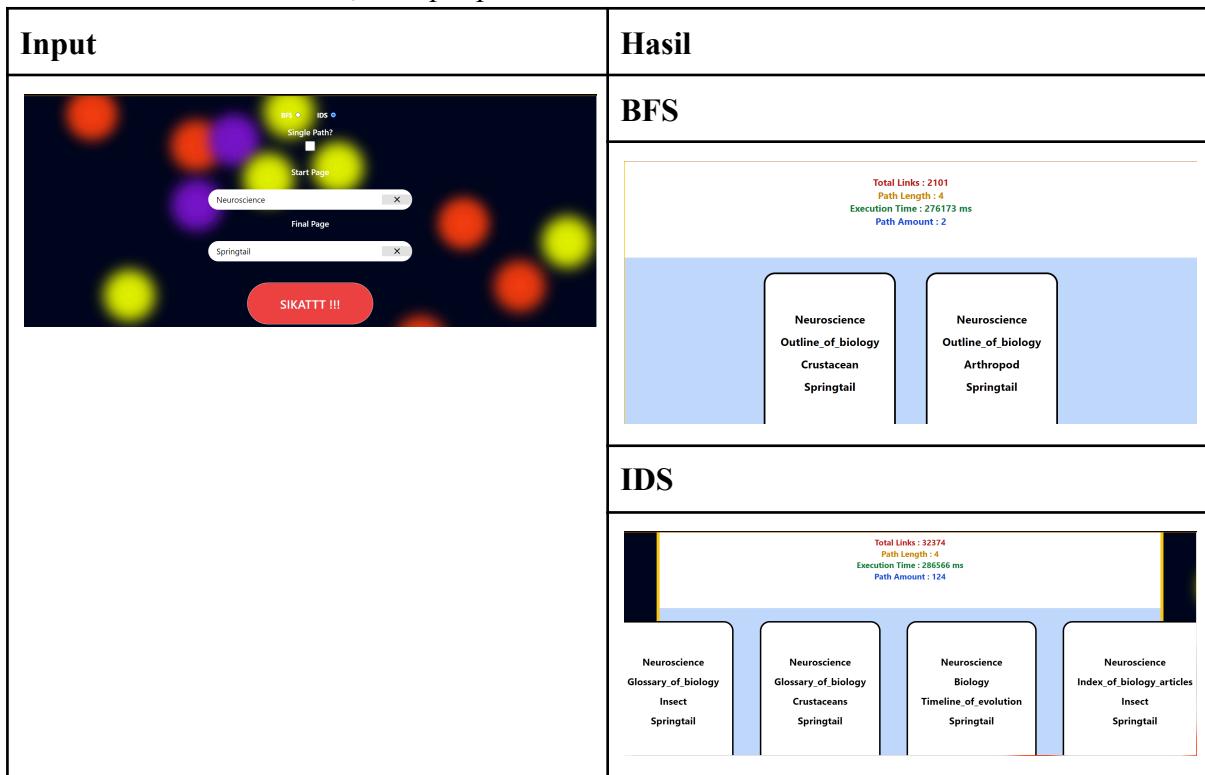
Input	Hasil
	<p><b>BFS</b></p> <p><b>RESULTS</b></p> <ul style="list-style-type: none"> <li>Total Links : 351</li> <li>Path Length : 2</li> <li>Execution Time : 3248 ms</li> <li>Path Amount : 1</li> </ul>
	<p><b>IDS</b></p> <p><b>RESULTS</b></p> <ul style="list-style-type: none"> <li>Total Links : 1057</li> <li>Path Length : 2</li> <li>Execution Time : 9105 ms</li> <li>Path Amount : 1</li> </ul>

### 4. Kasus kedalaman 2, *multiple path*

Input	Hasil
-------	-------



### 5. Kasus kedalaman 3, *multiple path*



## D. Analisis Hasil Pengujian

Berdasarkan hasil pengujian di atas, algoritma memberikan hasil dengan eksekusi waktu yang lebih cepat. Hal ini mungkin disebabkan karena implementasi konkurensi yang berbeda dari kedua algoritma tersebut. Algoritma BFS membatasi jumlah konkurensi pada proses *scraping* dalam *queue* sehingga batas konkurensi bersifat tetap atau konstan. Sedangkan IDS membatasi penambahan konkurensi pada setiap anak-anak yang akan dibangkitkan dalam fungsi rekursif, sehingga jumlah konkurensi tidak tetap melainkan eksponensial. Tentu saja perlakuan tersebut dapat menghasilkan jumlah konkurensi yang tidak terkontrol dalam jangka waktu panjang, namun pada durasi waktu  $< 5$  menit, implementasi tersebut justru membuat proses IDS menjadi sangat cepat dan lebih efektif.

Terlihat pula pada pengujian nomor 5 dimana dengan judul awal dan akhir yang sama, algoritma IDS memberikan hasil *multiple path* yang lebih banyak dari pada hasil algoritma BFS. Hal ini merupakan contoh langsung dari pernyataan di atas dimana algoritma IDS yang kami implementasikan lebih cepat dari algoritma BFS.

Pada pengujian nomor 2 algoritma BFS, didapatkan hasil eksekusi sejumlah 0 ms. Hal ini terjadi karena sebelumnya, *test case* yang sama telah dilakukan oleh algoritma IDS sehingga data telah tersimpan dalam *cache*.

## **BAB V**

### **KESIMPULAN DAN SARAN**

Dari tugas besar kedua IF2211 Strategi Algoritma ini, kami berhasil membuat *WikiRace Solver* yang dapat digunakan untuk mencari jalur terdekat dari suatu laman Wikipedia ke laman Wikipedia lainnya. Dari hasil pengujian kami, dapat disimpulkan bahwa algoritma IDS memiliki performa yang lebih baik daripada algoritma BFS karena algoritma IDS memiliki kompleksitas ruang yang lebih rendah dibandingkan BFS. Selain itu, algoritma BFS mengunjungi semua pranala yang ditemukan sehingga dapat memakan waktu yang lebih lama.

Saran untuk tugas besar ini adalah menggunakan target laman yang tidak bisa memblokir supaya pencarian dapat dilakukan dengan lebih cepat dan juga tidak memberatkan perangkat.

## **LAMPIRAN**

### **GitHub**

Kode dapat diakses pada repository GitHub

[https://github.com/Edvardesu/Tubes2\\_Dominus-Vobiscum](https://github.com/Edvardesu/Tubes2_Dominus-Vobiscum)

Jika terdapat masalah, dapat menghubungi ID Line : vansonk

## **DAFTAR PUSTAKA**

Munir, R. (2021). *Breadth/Depth First Search (BFS-DFS) Bagian 1*. Diakses pada 27 April 2024, dari  
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf>

Munir, R. (2021). *Breadth/Depth First Search (BFS-DFS) Bagian 2*. Diakses pada 27 April 2024, dari  
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf>

Anonimus. (2023). *Web Scraping in Golang: Complete Guide 2024*. Diakses pada 22 April 2024, dari <https://www.zenrows.com/blog/web-scraping-golang#how-to-web-scrape-in-go>