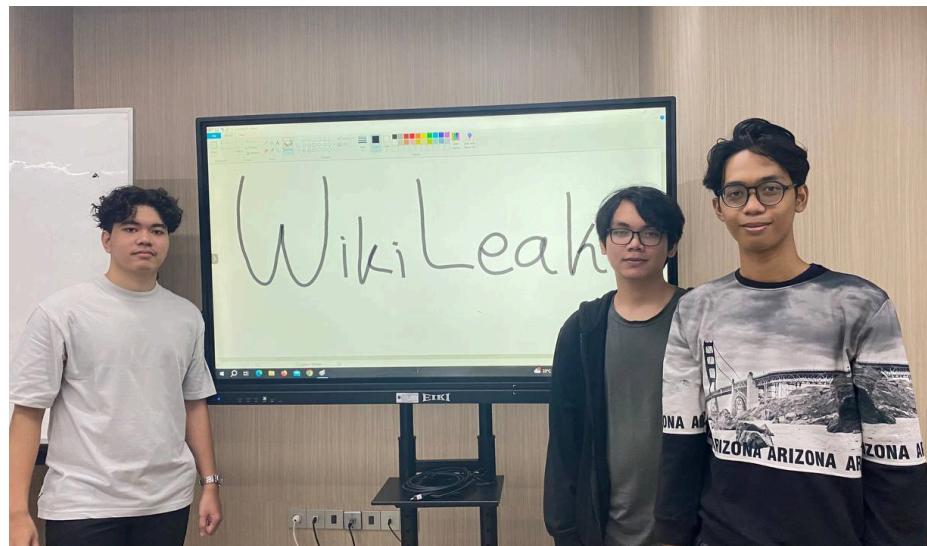


**LAPORAN TUGAS BESAR II**  
**IF2211 STRATEGI ALGORITMA**  
**“Pemanfaatan Algoritma IDS dan BFS**  
**dalam Permainan WikiRace”**



**Dosen:**

Ir. Rila Mandala, M. Eng, Ph. D.  
Monterico Adrian, S. T., M. T

**WikiLeak**

**Kelompok 58:**

13522131 Owen Tobias Sinurat  
13522138 Andi Marihot Sitorus  
13522150 Albert Ghazaly

**PROGRAM STUDI TEKNIK INFORMATIKA**  
**INSTITUT TEKNOLOGI BANDUNG**  
**SEMESTER II TAHUN 2023/2024**

# **BAB I**

## **DESKRIPSI TUGAS**

### **1.1. 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.

### **1.2. Spesifikasi Tugas**

Berikut adalah spesifikasi tugas yang diberikan.

- Buatlah program dalam bahasa Go yang mengimplementasikan algoritma IDS dan BFS untuk menyelesaikan permainan WikiRace.
- Program menerima masukan berupa jenis algoritma, judul artikel awal, dan judul artikel tujuan.
- Program memberikan keluaran berupa jumlah artikel yang diperiksa, jumlah artikel yang dilalui, rute penjelajahan artikel (dari artikel awal hingga artikel tujuan), dan waktu pencarian (dalam ms).
- Program cukup mengeluarkan salah satu rute terpendek saja (cukup satu rute saja, tidak perlu seluruh rute kecuali mengerjakan bonus).
- Program berbasis web, sehingga perlu dibuat front-end dan back-end (tidak perlu di-deploy).
- Repository front-end dan back-end diizinkan untuk dipisah maupun digabung dalam repository yang sama.
- Program wajib dapat mencari rute terpendek kurang dari 5 menit untuk setiap permainan.
- Tugas dikerjakan berkelompok dengan anggota minimal 2 orang dan maksimal 3 orang, boleh lintas kelas dan lintas kampus. Akan tetapi, anggota kelompok tidak boleh sama dengan anggota kelompok pada tugas-tugas Strategi Algoritma sebelumnya.
- Program harus mengandung komentar yang jelas serta mudah dibaca.

- Mahasiswa dilarang menggunakan kode program yang didapatkan dari internet (alasan menggunakan kakas seperti GitHub Copilot tidak diterima). Mahasiswa harus membuat program sendiri, diperbolehkan untuk belajar dari program yang sudah ada.
- Jika terdapat kesulitan selama mengerjakan tugas besar sehingga memerlukan bimbingan, maka dapat melakukan asistensi tugas besar kepada asisten (opsional). Dengan catatan asistensi hanya bersifat membimbing, bukan memberikan “jawaban”.
- Terdapat juga demo dari program yang telah dibuat. Pengumuman tentang demo menunggu pemberitahuan lebih lanjut dari asisten.
- Setiap kelompok harap mengisi nama kelompok dan anggotanya pada pranala [bit.ly/kelompoktubes2stima24](https://bit.ly/kelompoktubes2stima24), paling lambat Kamis, 4 April pukul 22.11 WIB.
- Diwajibkan untuk memilih asisten meskipun tidak melakukan asistensi, karena asisten yang dipilih akan menjadi asisten saat asistensi (opsional) dan demo tugas besar. Pemilihan asisten dapat dilakukan pada link berikut, paling lambat Kamis, 4 April pukul 22.11 WIB.
  - Program disimpan dalam repository yang bernama Tubes2\_NamaKelompok (bila digabung) dan Tubes2\_FE/BE\_NamaKelompok (bila dipisah) dengan nama kelompok sesuai dengan yang di sheets diatas. Berikut merupakan struktur dari isi repository tersebut:
    - a. Folder src berisi program yang dapat dijalankan IF2211 Strategi Algoritma – Tugas Besar 1 2
    - b. Folder doc berisi laporan tugas besar dengan format NamaKelompok.pdf
    - c. README untuk tata cara penggunaan yang minimal berisi:
      - i. Penjelasan singkat algoritma IDS dan BFS yang diimplementasikan
      - ii. Requirement program dan instalasi tertentu bila ada
      - iii. Command atau langkah-langkah dalam meng-compile atau build program
      - iv. Author (identitas pembuat)
  - Laporan dikumpulkan hari Sabtu, 27 April 2024 pada alamat Google Form berikut paling lambat pukul 23.59 : <https://bit.ly/tubes2stima24>
  - Adapun pertanyaan terkait tugas besar ini bisa disampaikan melalui QnA berikut: <https://bit.ly/qnastima24>

## **BAB II**

### **LANDASAN TEORI**

#### **2.1 Dasar Teori**

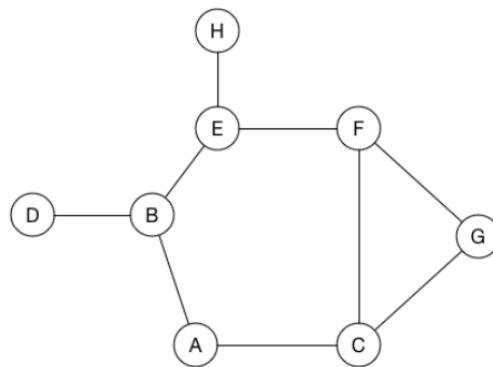
##### **2.1.1. Penjelajahan Graf**

Penjelajahan graf adalah proses traversal atau perjalanan melalui semua simpul (node) dalam sebuah graf. Tujuan dari penjelajahan graf adalah untuk mengunjungi setiap simpul dalam graf tepat satu kali. Penjelajahan graf sering digunakan dalam pemrosesan dan analisis graf, yang meliputi pencarian jalur, pemetaan jaringan, algoritma pencarian jarak terpendek, dan berbagai aplikasi lainnya.

Ada beberapa algoritma yang umum digunakan untuk melakukan penjelajahan graf, di antaranya: Breadth First Search (BFS), Depth First Search (DFS), dan Iterative Depth Search (IDS).

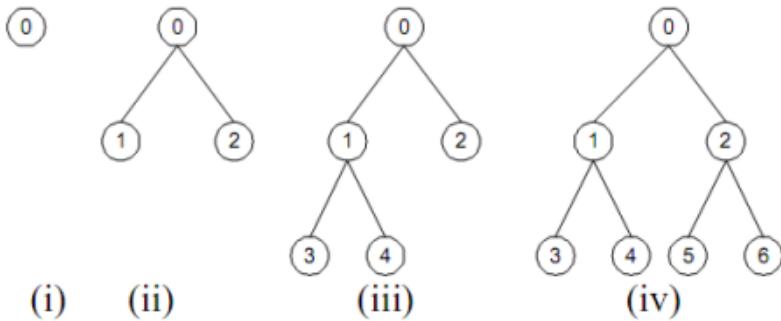
##### **2.1.2. Breadth First Search (BFS)**

Algoritma BFS melakukan pencarian melebar dengan mengunjungi node pada level n terlebih dahulu sebelum mengunjungi node-node pada level n+1.



Urutan simpul-simpul yang dikunjungi secara BFS dari A → A, B, C, D, E, F, G, H

Agar lebih jelas, berikut adalah contoh rute pencarian BFS dalam bentuk tree. Pencarian dimulai dari angka nol dan naik satu-persatu sesuai urutan.

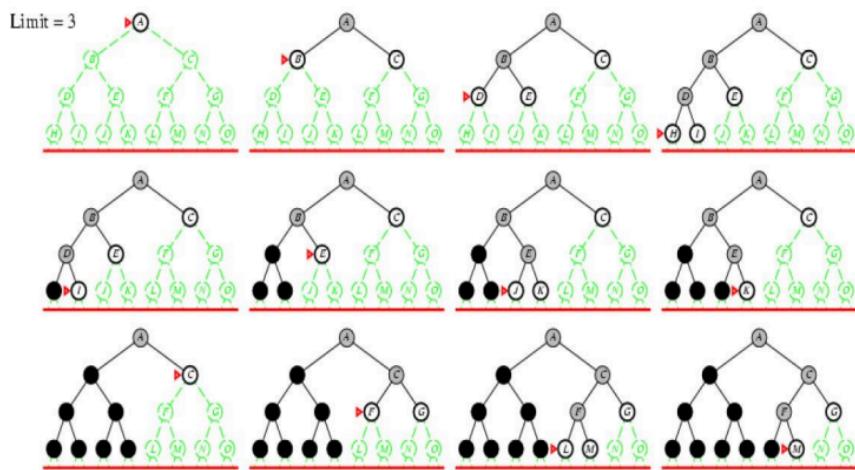


### 2.1.3. Iterative Deepening Search (IDS)

IDS merupakan kombinasi dari DFS dan BFS. Algoritma ini melakukan DFS dengan batasan kedalaman (depth limit) yang ditingkatkan secara iteratif hingga menemukan solusi. IDS berguna ketika kedalaman graf tidak diketahui dan memungkinkan pencarian solusi dengan menggunakan memori yang terbatas.

Berikut adalah contoh IDS pada tree dengan kedalaman 3

### IDS dengan $d=3$



## 2.2 Cara Kerja Program Secara Umum

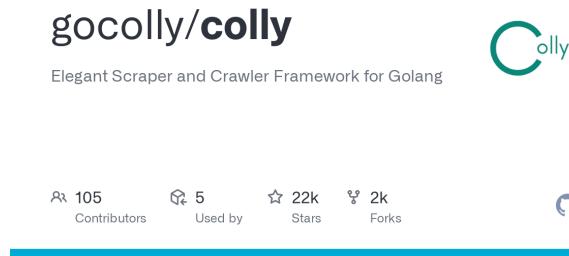
Program akan mencari rute terpendek antara dua artikel wikipedia dengan *web scraping*. Rute ini didapatkan dengan cara menelusuri link yang berada di artikel awal, lalu menelusuri link yang berada dalam link-link sebelumnya hingga mencapai artikel tujuan. Program ini menggunakan konsep tree untuk menyimpan link yang dihasilkan dari *web scrap* sehingga dapat memberikan rute dari artikel awal ke artikel tujuan dengan mencari link artikel tujuan.

Program menelusuri tree dengan dua algoritma penelusuran graf yaitu Breadth First Search (BFS) dan Iterative Deepening Search (IDS). Program ini berbasis web dan meminta masukan berupa judul dari artikel awal dan tujuan serta jenis algoritma yang digunakan dalam pencarian rute. Program akan menampilkan hasil berupa rute terpendek, kedalaman atau level artikel tujuan, jumlah artikel yang ditelusuri, dan waktu yang dibutuhkan untuk menelusuri link.

## BAB III

# APLIKASI ALGORITMA SEARCHING

### 3.1 Strategi Web Scraping



Gambar 3.1.1 Go Colly

Pada tugas ini, kami menggunakan struktur data berupa tree untuk merepresentasikan sebuah *website* yang menyimpan hal-hal penting seperti nama link, parent, dan children. Scraping pada tugas besar ini menggunakan tools berupa Go Colly, yakni sebuah *web scraping framework* untuk *Golang*. Selain menggunakan *web scraping tools* seperti Go Colly, pada tugas besar ini, digunakan juga *tools* multiprocessing agar dapat mengekstrak link pada website secara sinkronus. *tools* yang digunakan adalah *Sync Wait Group* untuk me-manage penggunaan array hasil ekstrak agar tidak terjadinya *race condition*. Selain itu, kita juga menggunakan *Go Colly Queue*, yakni sebuah *tools* untuk mengekstrak *link* dari beberapa *website* secara sinkronus yang *go routine* nya sudah diatur langsung oleh Go Colly. Di samping itu, cache juga digunakan berupa *global variable*. Terdapat dua variabel cache berbeda yang digunakan, yakni cache 1 berupa map (key: link, value: bool) yang digunakan untuk menyimpan apakah sebuah website sudah dikunjungi atau belum untuk tidak memperbolehkan website duplikat dan cache 2 berupa map (key: link, value: list of node (children dari link pada key)) untuk *load* pada *scraping* selanjutnya agar memudahkan proses.

```
// def
type TreeNode struct {
    Title      string
    Link       string
    Parent     *TreeNode
    Children   []*TreeNode
    id         int
    imagePath string
}

// ctor
func NewTreeNode(title string, link string) *TreeNode {
    return &TreeNode{
        Title:      title,
        Link:       link,
        Parent:     nil,
        Children:   []*TreeNode{},
        id:         0,
        imagePath: "",
    }
}
```

Gambar 3.1.2 Struktur data tree

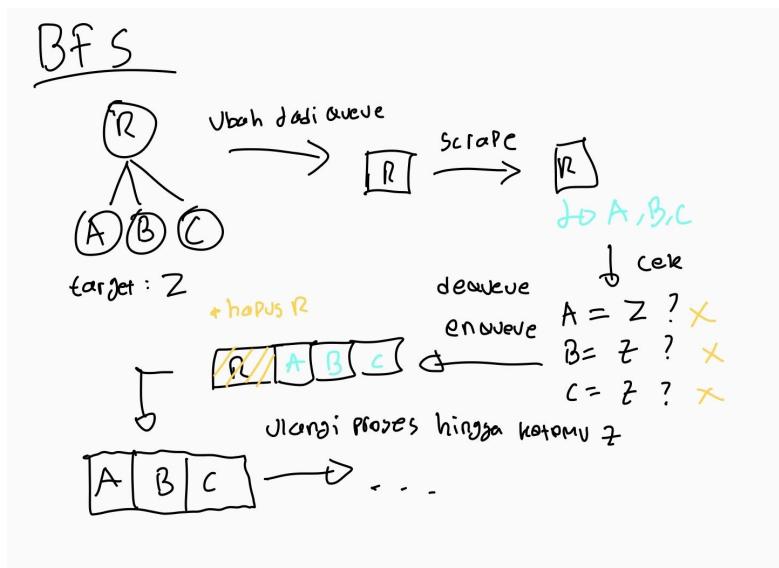
```
4 func ScrapeLink(node *TreeNode, target string, cache *Cache) {
5     // if node.Parent != nil {
6     //     fmt.Println("Scrape : ", node.Parent.Link, " ", node.Link, " ", node.id)
7     // } else {
8     //     fmt.Println("Scrape : ", node.Link, " ", node.id)
9     //
10    if node.Link[0:6] == "/wiki/" {
11        res, err := http.Get("https://en.wikipedia.org" + node.Link)
12        if err != nil {
13            log.Fatal(err)
14        }
15        defer res.Body.Close()
16    }
17 }
```

Gambar 3.1.3 Fungsi scraping

Berikut merupakan langkah-langkah secara ringkas dari strategi *scraping* yang digunakan

1. Validasi link yang *di-input*, yakni sebuah *node* apakah sudah sesuai dengan format (/wiki/....)
2. Cek apakah link tersebut sudah ada di cache 2, jika sudah maka *load* dari *cache* tersebut tanpa melanjutkan *scraping*
3. Jika tidak, ambil title dari web dan masukan dalam node link yang *di-scape*
4. Simpan link yang *di-scrape* pada cache 1 dengan menjadikan nilai dengan key = link tersebut menjadi bernilai *true*
5. Ambil *hyperlink* pada website lalu validasi menggunakan beberapa kriteria
  - a. Buang link yang tidak berawalan “/wiki/” karena dipastikan bukan artikel wikipedia
  - b. Buang link yang menuju Main\_Page karena link tersebut tidak konsisten (berubah-ubah)
  - c. Buang link yang terdapat “#” dan “.” karena link tersebut tidak berisi artikel
  - d. Buang link yang sudah pernah diekstrak (duplikat) dengan menggunakan cache 1 dan mengakses menggunakan link tersebut.
  - e. Buang link children yang sama dengan link yang *di-scrape* (*parent*)
6. Simpan *children link* tersebut pada atribut *children* dari *node* yang *di-scrape*
7. Simpan link dan children link pada cache 2 agar dapat *di-load* pada proses *search* selanjutnya

## 3.2 Algoritma Pencarian BFS



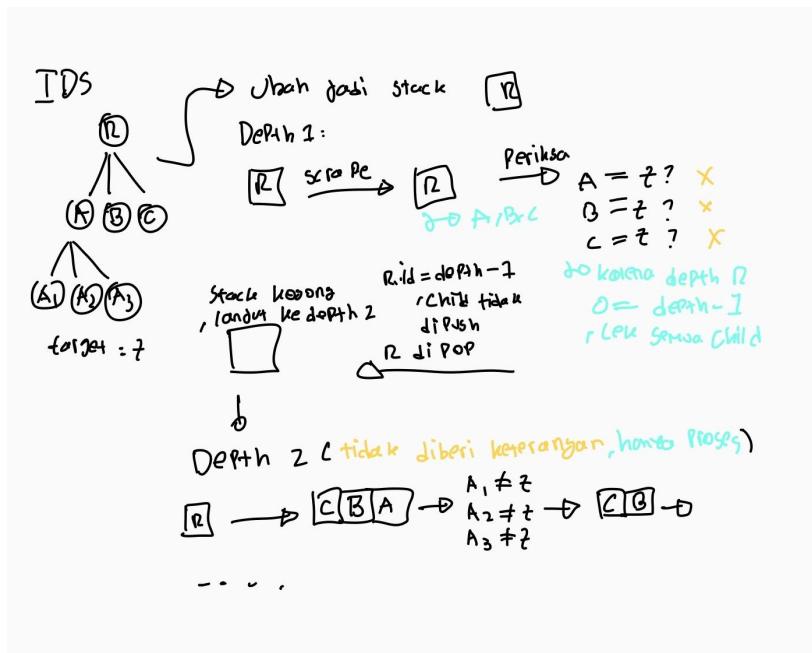
Gambar 3.2 Ilustrasi algoritma BFS

Algoritma pencarian BFS atau *Breadth First Search Algorithm* yang digunakan pada program ini menggunakan konsep *Queue* dalam merepresentasikan *Tree* yakni *Queue* tersebut berisi *pointer* dari *tree node*. Jadi, *queue* tersebut diiterasi dengan cara di-*dequeue* elemen paling awal, di-*scrape*, dan di-*enqueue* serta diperiksa *children* dari elemen paling awal tersebut apakah merupakan target yang diinginkan. Karena digunakan *multiprocessing*, maka proses *scrapping* dilakukan secara sinkronus sebanyak  $\min(\text{len}(\text{queue}), 150)$  agar mencegah *out of range error*. Selain itu, proses pengecekan dilakukan setelah proses sinkronus *web scraping* usai agar mencegah *race condition*. Berikut merupakan langkah-langkah ringkas dari algoritma pencarian BFS

1. Buat sebuah queue yang berisi *root node* yakni node yang menyimpan path awal dan variabel *found* yang berfungsi sebagai *flag* apakah program sudah menemukan link target.
2. *Loop* selama target belum ketemu (*found* bernilai *false*) lakukan langkah-langkah berikut
3. Jika queue hanya berisi *root* (kondisi awal), maka *scrap root* tersebut lalu periksa setiap *children*. Lalu, *dequeue* *root* dari *queue* dan *enqueue* *children* ke *queue*

4. Jika queue berisi banyak *node* (kondisi normal), maka *scrap node* sebanyak  $\min(150, \text{len(queue)})$  secara sinkronus dengan memanggil prosedur *scrapeLink()*
5. Pastikan proses *scrap* telah usai dengan menggunakan sync.WaitGroup (*tools* dari Go)
6. Periksa setiap *children* dari elemen *queue* sebanyak  $\min(150, \text{len(queue)})$  elemen, apakah salah satunya merupakan *link target*
7. Jika ketemu, ketemu return node tersebut
8. Jika belum, maka dequeue *queue* sebanyak  $\min(100, \text{len(queue)})$  elemen dan lakukan iterasi dimulai dari langkah 2

### 3.3 Algoritma Pencarian IDS



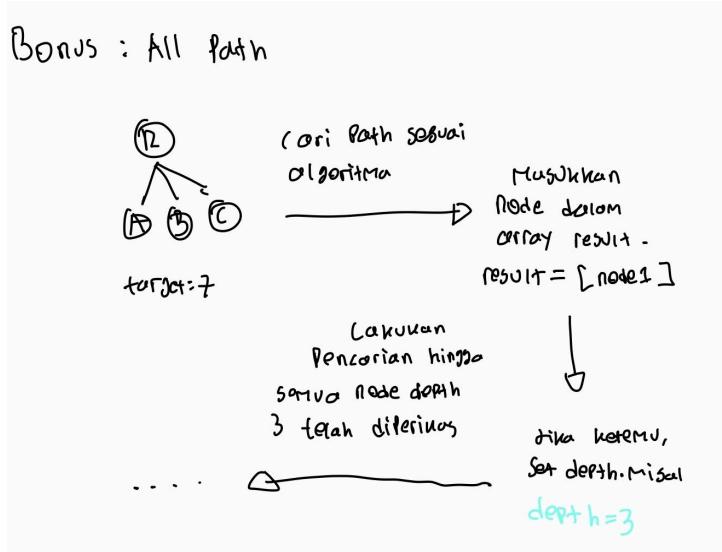
Gambar 3.3 Ilustrasi algoritma IDS

Algoritma pencarian IDS atau *Iterative Deepening Search Algorithm* yang digunakan menggunakan konsep *stack* dalam merepresentasikan *Tree* yakni *stack* tersebut menyimpan *pointer* dari *tree node*. Algoritma ini berjalan dengan *pop* elemen akhir, *scrape* elemen akhir tersebut, dan periksa *children* dari elemen tersebut apakah merupakan target link serta *push children* dalam *stack* jika bukan. Karena digunakan multiprocessing, maka proses scrapping

dilakukan secara sinkronus sebanyak  $\min(\text{len}(\text{stack}), 150)$  agar mencegah out of range error. Selain itu, proses pengecekan dilakukan setelah proses sinkronus web scraping usai agar mencegah race condition. Berikut merupakan langkah-langkah ringkas dari algoritma pencarian IDS

1. Buat sebuah *stack* yang berisi *root node* yakni node yang menyimpan path awal dan variabel *found* yang berfungsi sebagai *flag* apakah program sudah menemukan link target.
2. *Loop* selama target belum ketemu (*found* bernilai *false*) lakukan langkah-langkah berikut dan buat variabel *i* dengan nilai 1 dan akan terus ditambah sebagai penanda depth
3. buat variabel *stack* yang berisi *root node* lalu buatlah loop tambahan yang akan berhenti ketika *stack kosong*. Dilakukan proses berikut selama loop 2 berjalan
4. Jika *stack* hanya berisi *root* (kondisi awal), maka *scrape* elemen tersebut, periksa apakah *children node* berisi link target. Jika iya return *node*, jika tidak *pop root* dan *push children node* ke dalam *stack*
5. Jika *stack* tidak hanya berisi *root* (kondisi normal), maka cek sebanyak  $\min(\text{len}(\text{queue}), 150)$  dari elemen paling akhir dari *stack* apakah merupakan link target.
6. Jika, tidak maka *scrape* semua nya jika belum di-*scrape*. Selanjutnya, periksa jika link yang di-*scrape* memiliki kedalaman depth-1 (merupakan node yang memiliki *children* paling ujung). Jika iya maka periksa *children* nya dan apabila tidak ada link target, maka *children* dari *node* tidak perlu dimasukkan ke dalam *stack*
7. Jika, link yang di-*scrape* bukan memiliki kedalaman depth-1 (bukan merupakan node yang memiliki *children* paling ujung), maka periksa *children*-nya dan apabila tidak ada link target, maka *children* dari node akan masuk ke dalam *stack*
8. pop elemen *stack* sebanyak  $\min(\text{len}(\text{stack}), 150)$  dan *push children* dari *stack* sesuai langkah 6 dan 7
9. Lakukan langkah-langkah dari langkah 2 hingga iterasi selesai dan link target ketemu

### 3.4 Algoritma Pencarian All Path



Gambar 3.4 Ilustrasi algoritma All Path

Algoritma all path bisa dibilang cukup sama untuk *all path BFS* maupun *all path IDS*, yang membedakan mereka berdua adalah algoritma pencarinya saja. Inti dari pencarian *All Path* adalah ketika mendapatkan resul, tidak langsung return dan selesai, tetapi masukkan hasil ke dalam sebuah array result dan catat kedalaman result. Nantinya, akan diiterasi hingga semua *node* yang memiliki kedalaman sama dengan result habis maka akan direturn array result. Berikut merupakan langkah-langkah ringkas mengenai *all path*

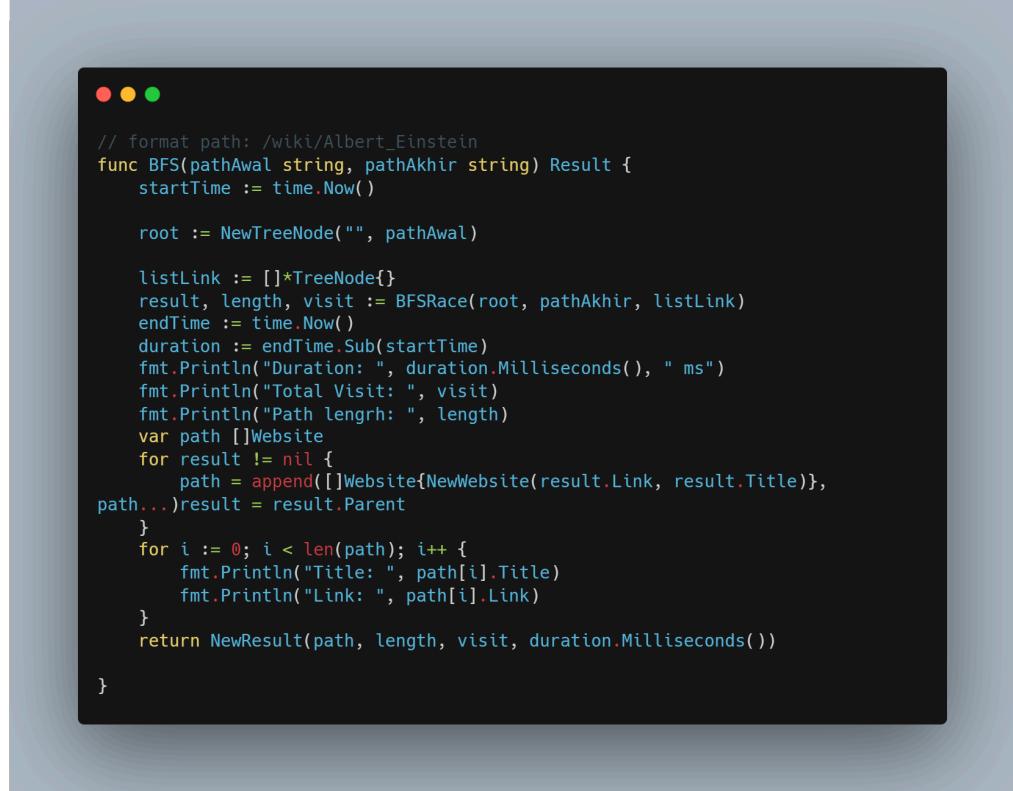
1. Buatlah variabel depth dengan nilai awal -1, dan array kosong result
2. buatlah loop yang hanya berhenti ketika nilai variabel depth bukan -1 **dan** nilai depth pada queue/stack paling kecil lebih besar dari variable depth di atas.
3. lakukan algoritma yang sama sesuai dengan pilihan (BFS atau IDS), dengan cara yang telah dijelaskan pada 3.2 dan 3.3.
4. Ketika terdapat node dengan link yang sama dengan target maka diubah nilai variabel depth dari -1 menjadi kedalaman *node* dan node dimasukkan ke dalam array result
5. lakukan hal yang sama hingga *loop* usai

## BAB IV

### IMPLEMENTASI DAN PENGUJIAN

#### 4.1 Spesifikasi Teknis Program

Berikut adalah beberapa implementasi fungsi untuk website WikiLeaks (hanya menampilkan fungsi krusial).

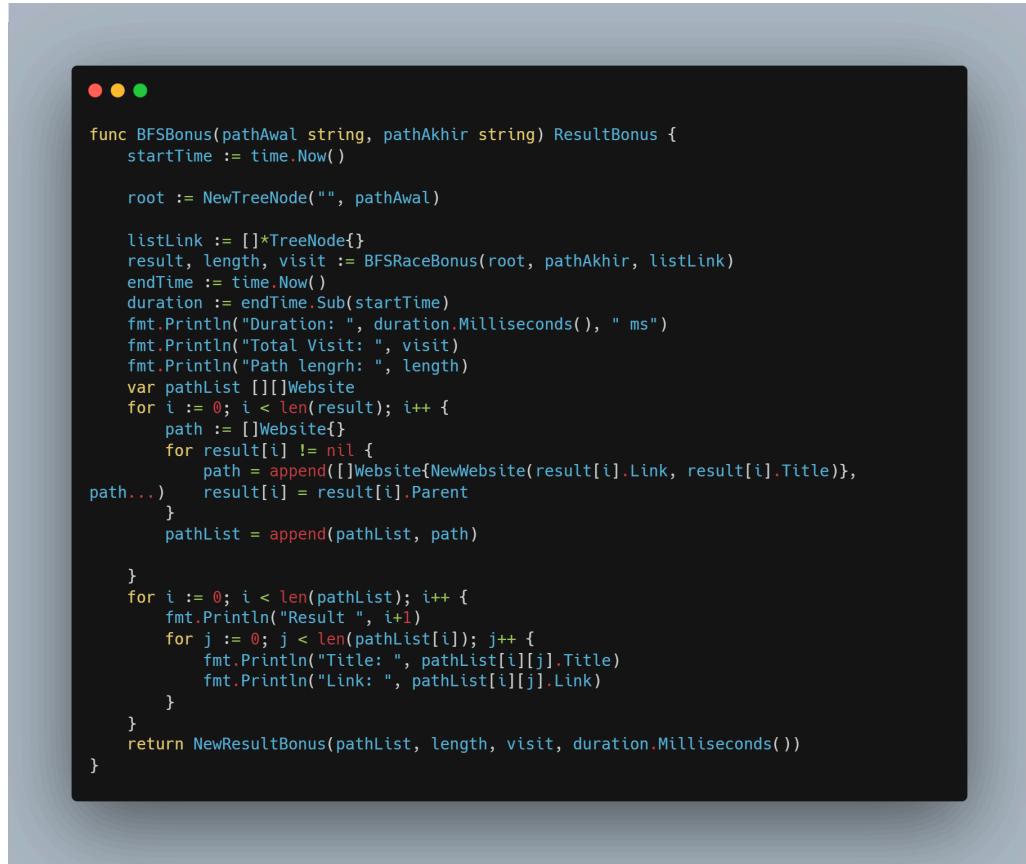


```
// format path: /wiki/Albert_Einstein
func BFS(pathAwal string, pathAkhir string) Result {
    startTime := time.Now()

    root := NewTreeNode("", pathAwal)

    listLink := []*TreeNode{}
    result, length, visit := BFSSearch(root, pathAkhir, listLink)
    endTime := time.Now()
    duration := endTime.Sub(startTime)
    fmt.Println("Duration: ", duration.Milliseconds(), " ms")
    fmt.Println("Total Visit: ", visit)
    fmt.Println("Path length: ", length)
    var path []Website
    for result != nil {
        path = append([]Website{NewWebsite(result.Link, result.Title)}, path...)
        result = result.Parent
    }
    for i := 0; i < len(path); i++ {
        fmt.Println("Title: ", path[i].Title)
        fmt.Println("Link: ", path[i].Link)
    }
    return NewResult(path, length, visit, duration.Milliseconds())
}
```

Gambar 1. Fungsi BFS untuk satu solusi

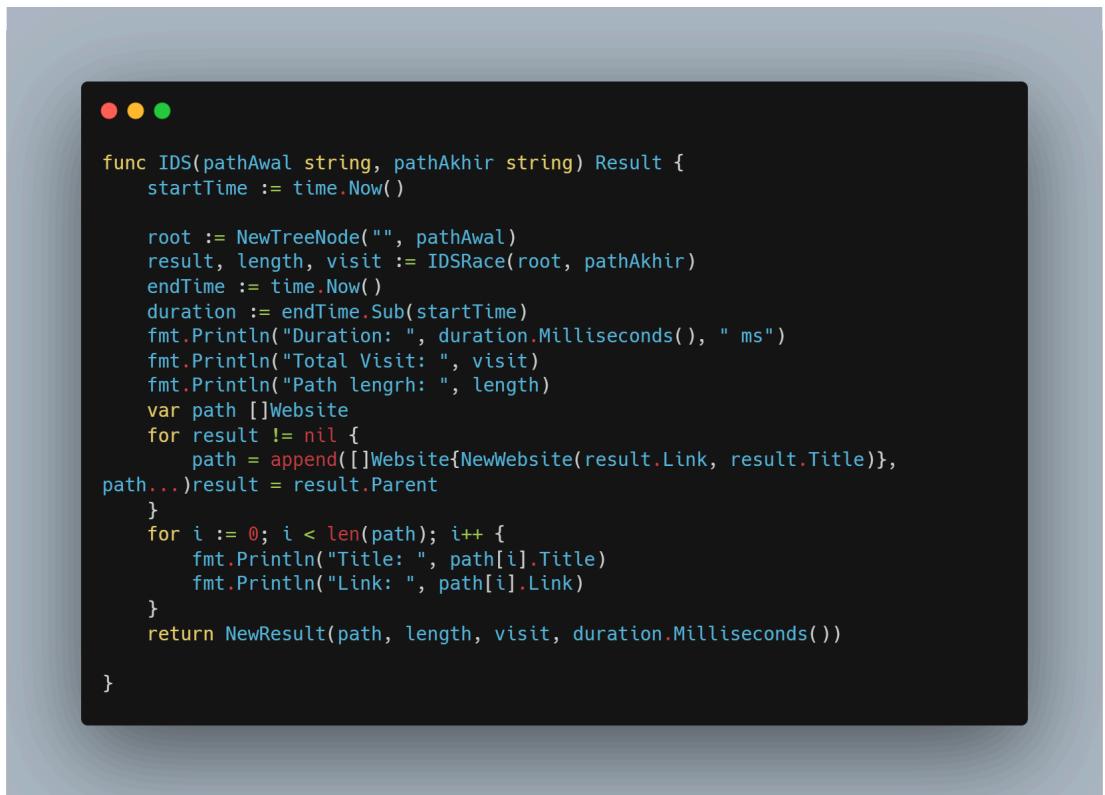


```
func BFSBonus(pathAwal string, pathAkhir string) ResultBonus {
    startTime := time.Now()

    root := NewTreeNode("", pathAwal)

    listLink := []*TreeNode{}
    result, length, visit := BFRaceBonus(root, pathAkhir, listLink)
    endTime := time.Now()
    duration := endTime.Sub(startTime)
    fmt.Println("Duration: ", duration.Milliseconds(), " ms")
    fmt.Println("Total Visit: ", visit)
    fmt.Println("Path length: ", length)
    var pathList [][]Website
    for i := 0; i < len(result); i++ {
        path := []Website{}
        for result[i] != nil {
            path = append([]Website{NewWebsite(result[i].Link, result[i].Title)}, path...)
            result[i] = result[i].Parent
        }
        pathList = append(pathList, path)
    }
    for i := 0; i < len(pathList); i++ {
        fmt.Println("Result ", i+1)
        for j := 0; j < len(pathList[i]); j++ {
            fmt.Println("Title: ", pathList[i][j].Title)
            fmt.Println("Link: ", pathList[i][j].Link)
        }
    }
    return NewResultBonus(pathList, length, visit, duration.Milliseconds())
}
```

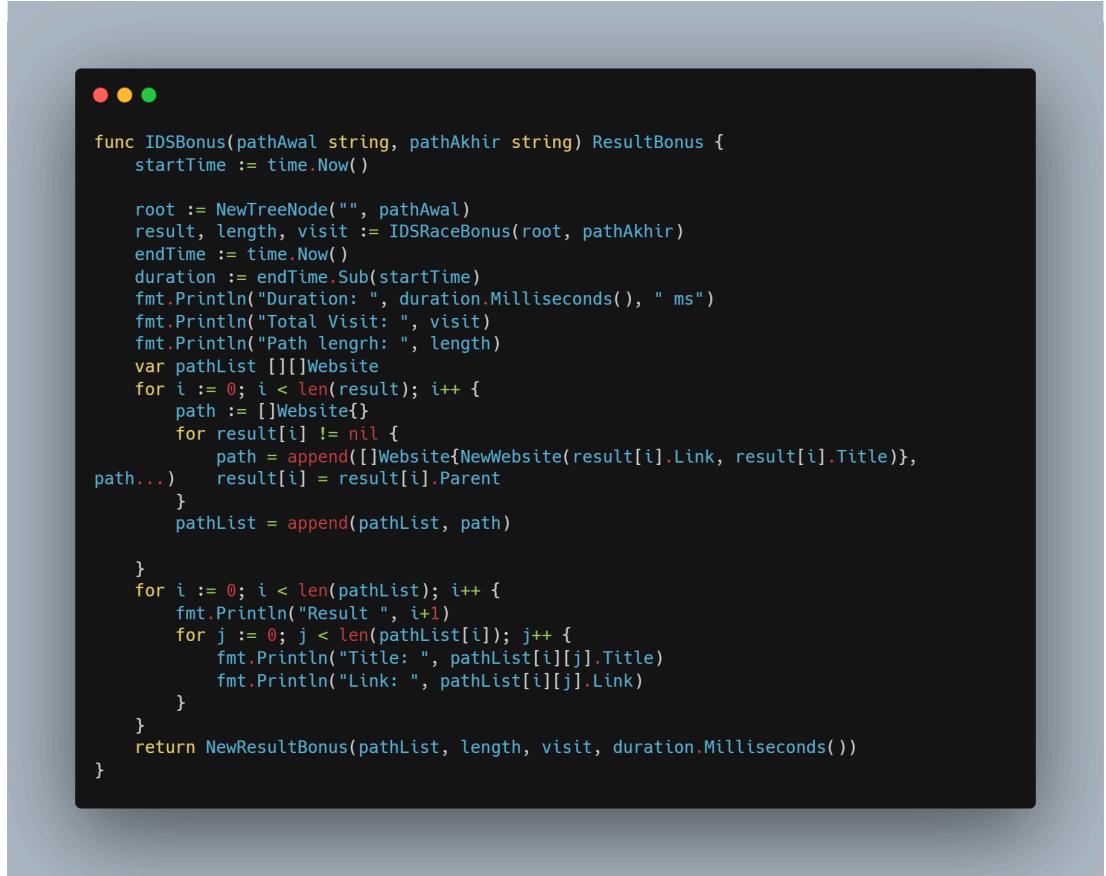
Gambar 2. Fungsi BFS untuk semua solusi yang mungkin



```
func IDS(pathAwal string, pathAkhir string) Result {
    startTime := time.Now()

    root := NewTreeNode("", pathAwal)
    result, length, visit := IDSRace(root, pathAkhir)
    endTime := time.Now()
    duration := endTime.Sub(startTime)
    fmt.Println("Duration: ", duration.Milliseconds(), " ms")
    fmt.Println("Total Visit: ", visit)
    fmt.Println("Path length: ", length)
    var path []Website
    for result != nil {
        path = append([]Website{NewWebsite(result.Link, result.Title)}, path...)
        result = result.Parent
    }
    for i := 0; i < len(path); i++ {
        fmt.Println("Title: ", path[i].Title)
        fmt.Println("Link: ", path[i].Link)
    }
    return NewResult(path, length, visit, duration.Milliseconds())
}
```

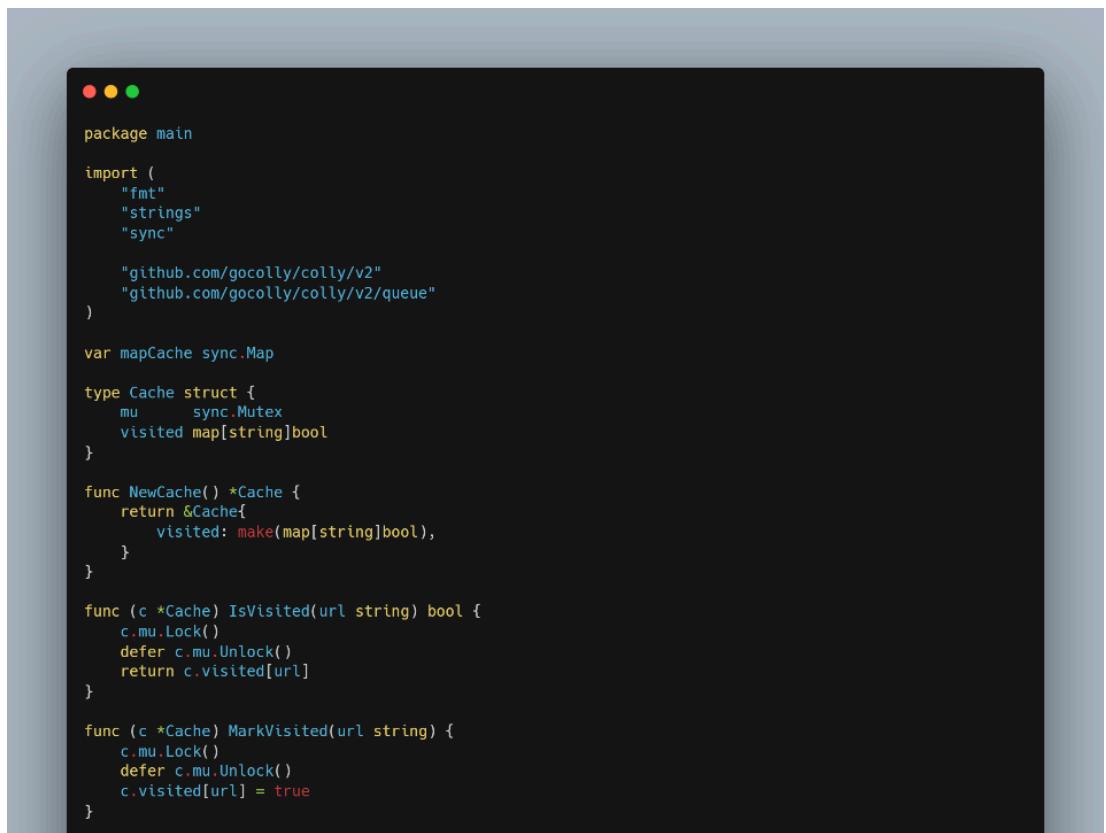
Gambar 3. Fungsi IDS untuk satu solusi



```
func IDSBonus(pathAwal string, pathAkhir string) ResultBonus {
    startTime := time.Now()

    root := NewTreeNode("", pathAwal)
    result, length, visit := IDSRaceBonus(root, pathAkhir)
    endTime := time.Now()
    duration := endTime.Sub(startTime)
    fmt.Println("Duration: ", duration.Milliseconds(), " ms")
    fmt.Println("Total Visit: ", visit)
    fmt.Println("Path lengrh: ", length)
    var pathList [][]Website
    for i := 0; i < len(result); i++ {
        path := []Website{}
        for result[i] != nil {
            path = append([]Website{NewWebsite(result[i].Link, result[i].Title)}, result...)
            result[i] = result[i].Parent
        }
        pathList = append(pathList, path)
    }
    for i := 0; i < len(pathList); i++ {
        fmt.Println("Result ", i+1)
        for j := 0; j < len(pathList[i]); j++ {
            fmt.Println("Title: ", pathList[i][j].Title)
            fmt.Println("Link: ", pathList[i][j].Link)
        }
    }
    return NewResultBonus(pathList, length, visit, duration.Milliseconds())
}
```

Gambar 4. Fungsi IDS untuk semua solusi yang mungkin



```
package main

import (
    "fmt"
    "strings"
    "sync"

    "github.com/gocolly/colly/v2"
    "github.com/gocolly/colly/v2/queue"
)

var mapCache sync.Map

type Cache struct {
    mu     sync.Mutex
    visited map[string]bool
}

func NewCache() *Cache {
    return &Cache{
        visited: make(map[string]bool),
    }
}

func (c *Cache) IsVisited(url string) bool {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.visited[url]
}

func (c *Cache) MarkVisited(url string) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.visited[url] = true
}
```

Gambar 5. Fungsi utilitas untuk membantu algoritma scraping

```
func ScrapeLink(node *TreeNode, target string, cache *Cache) {
    if node.Parent != nil {
        fmt.Println("Scrape : ", node.Parent.Link, " ", node.Link, " ", node.id)
    } else {
        fmt.Println("Scrape : ", node.Link, " ", node.id)
    }
    if node.Link[0:6] == "/wiki/" {

        if value, ok := mapCache.Load(node.Link); ok {
            if value != nil {
                for i := 0; i < len(value.([]*TreeNode)); i++ {
                    node.AddChild(NewTreeNode(value.([]*TreeNode)[i].Title, value.([]*TreeNode)[i].Link))
                }
            }
            return
        }
    }
    c := colly.NewCollector(
        colly.AllowedDomains("en.wikipedia.org"),
        // colly.Async(true),
    )
    q, _ := queue.New(
        15, // Number of consumer threads
        &queue.InMemoryQueueStorage{MaxSize: 200}, // Use in-memory queue storage
    )
    q.AddURL("https://en.wikipedia.org" + node.Link)

    // Define a callback function to be executed when a link is found
    c.OnHTML("h1#firstHeading", func(e *colly.HTMLElement) {
        // Extract text or any other attribute you want
        node.Title = strings.TrimSpace(e.DOM.Text())
    })
    c.OnHTML("a[href]", func(e *colly.HTMLElement) {
        // Extract the href attribute of the <a> element
        link := e.Attr("href")
        teks := e.Text
        // results <- link // Send the link to the results channel
        if !cache.IsVisited(link) {
            if link != target {
                cache.MarkVisited(link)
            }
            if len(link) >= 6 {
                if strings.Contains(link, "#") || strings.Contains(link, "Main_Page") ||
strings.Contains(link, ":") || link == node.Link {
                    // do nothing
                } else {
                    node.AddChild(NewTreeNode(teks, link))
                }
            }
        }
    })
}

// Define a callback function to be executed when the scraping is complete
c.OnScraped(func(r *colly.Response) {
    if _, ok := mapCache.Load(node.Link); !ok {
        mapCache.Store(node.Link, node.Children)
    }
})
q.Run(c)
}
```

Gambar 6. Fungsi utama algoritma scraping

```
● ● ● ●

package main

import (
    "fmt"
)

// def
type TreeNode struct {
    Title    string
    Link    string
    Parent   *TreeNode
    Children []*TreeNode
    id       int
}

// ctor
func NewTreeNode(title string, link string) *TreeNode {
    return &TreeNode{
        Title:    title,
        Link:    link,
        Parent:   nil,
        Children: []*TreeNode{},
        id:       0,
    }
}

// add children node
func (node *TreeNode) AddChild(child *TreeNode) {
    child.Parent = node
    child.id = node.id + 1
    node.Children = append(node.Children, child)
}

// get children num
func (node *TreeNode) GetNumberOfNodes() int {
    count := 1
    for _, child := range node.Children {
        count += child.GetNumberOfNodes()
    }
    return count
}

func (node *TreeNode) GetNumberOfChildren() int {
    return len(node.Children)
}

// print node (for debug)
func (node *TreeNode) PrintNode(indentation int) {
    fmt.Print(node.Title)
    fmt.Print(" ")
    fmt.Print(node.Link)
    fmt.Println()

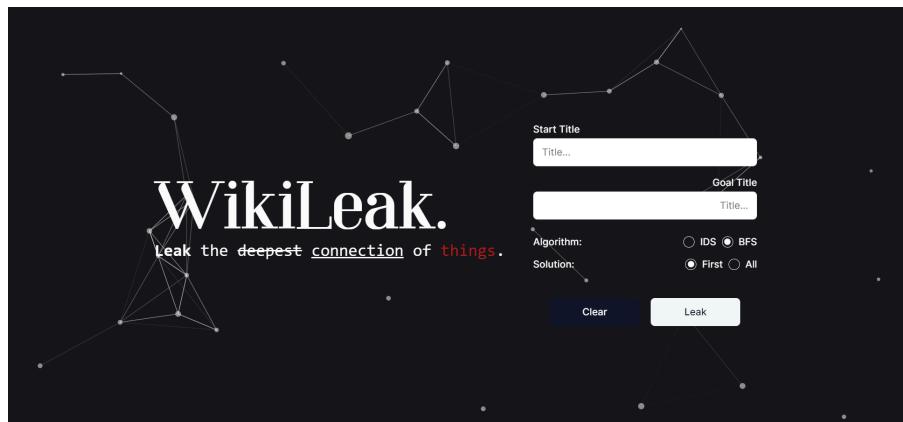
    for _, child := range node.Children {
        for i := 0; i < indentation; i++ {
            fmt.Print(" ")
        }
        child.PrintNode(indentation + 1)
    }
}
```

Gambar 7. Struktur data tree yang dipakai untuk mencari path

## 4.2 Tata Cara Penggunaan Program

Berikut adalah tata cara penggunaan website WikiLeaks:

1. *Clone repository* pada tautan:  
[https://github.com/owenthe10x/Tubes2\\_WikiLeaks](https://github.com/owenthe10x/Tubes2_WikiLeaks)
2. Buka terminal pada *repository*
3. Pindah ke directory “backend/src”
4. Jalankan “go build -o main.exe” untuk membangun program *backend*
5. Jalankan “./main.exe” untuk menjalankan server *backend*.
6. Buka terminal baru pada directory “frontend”.
7. Jalankan “npm install” untuk menginstal seluruh paket yang dibutuhkan.
8. Jalankan “npm run dev” untuk menjalankan *website*.
9. Buka *web browser* dan pergi ke ketikkan *url* “localhost:3000”.
10. Setelah *website* berhasil load, akan ditampilkan sebuah form yang harus diisi pengguna.



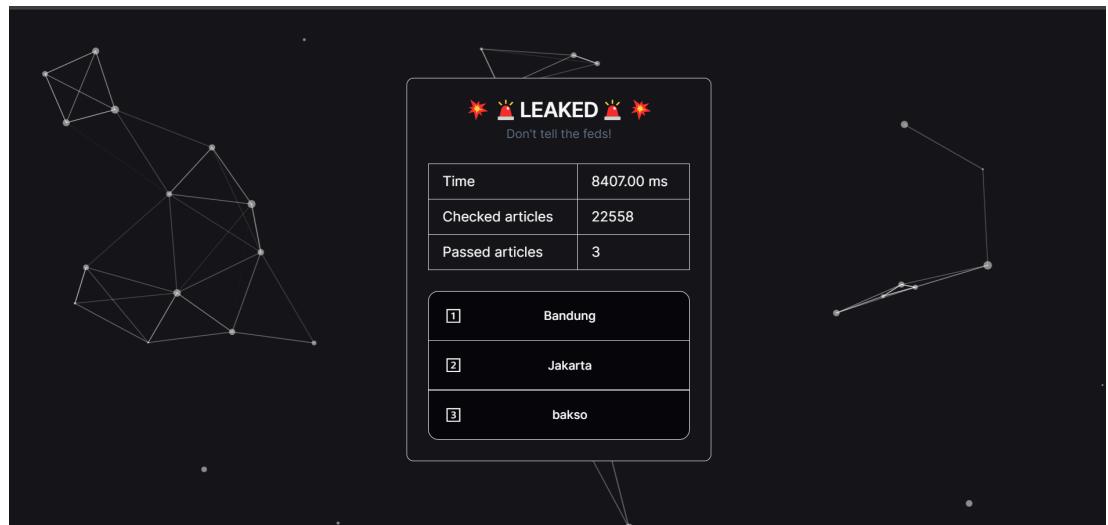
11. Isi sesuai ketentuan berikut:
  - a. Start Title : Judul artikel awal.
  - b. Goal Title : Judul artikel yang dituju.
  - c. Algorithm : Algoritma yang ingin digunakan untuk pencarian *path*, apakah IDS atau BFS.
  - d. Solution : Jumlah solusi yang diinginkan, apakah satu solusi tercepat atau semua solusi yang mungkin.
12. Klik “Leak” untuk mencari *path* sesuai input pada *form*.
13. Hasil akan muncul dalam waktu yang berbeda-beda bergantung pada *input* pada *form*.

### 4.3 Hasil Pengujian

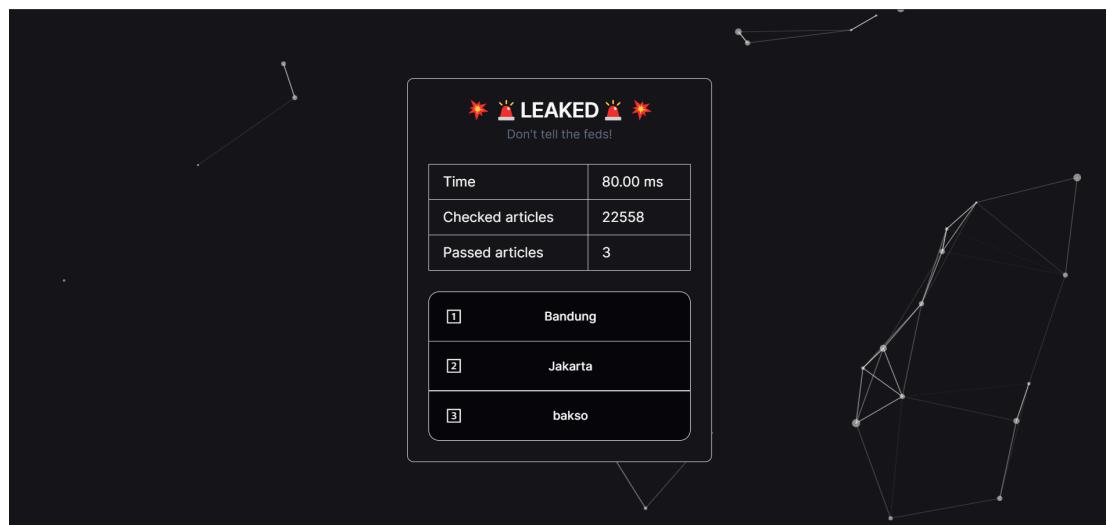
#### A. Pengujian 1

Judul Awal : Bandung

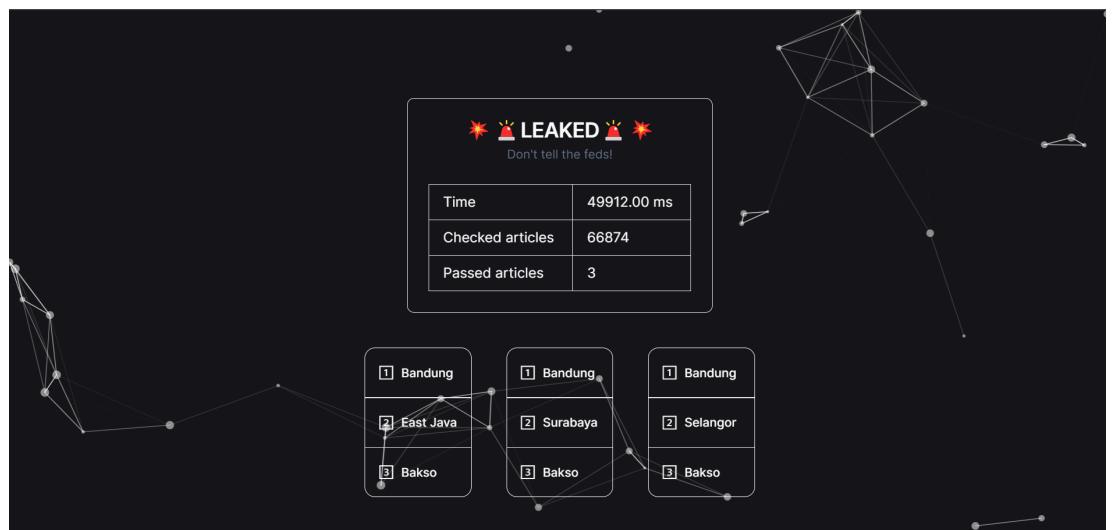
Judul Tujuan : Bakso



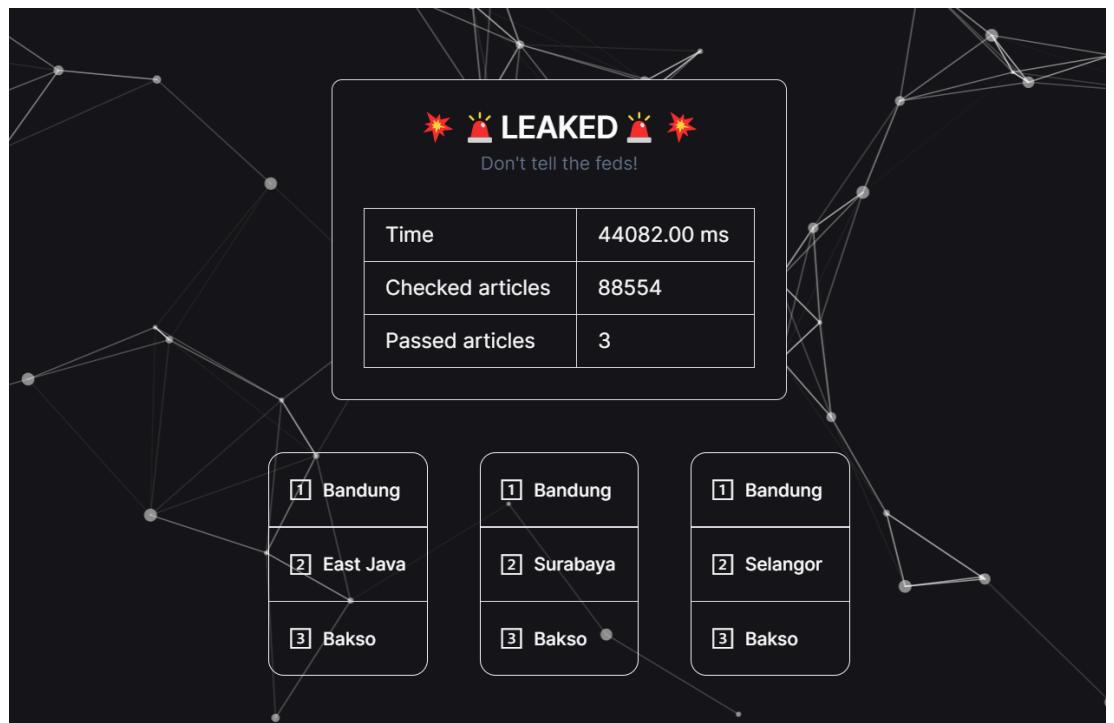
Gambar 1. BFS dengan Satu Solusi



Gambar 2. IDS dengan Satu Solusi dengan implementasi cache



Gambar 3. IDS dengan Semua Solusi yang Mungkin



Gambar 4. IDS dengan Semua Solusi yang Mungkin

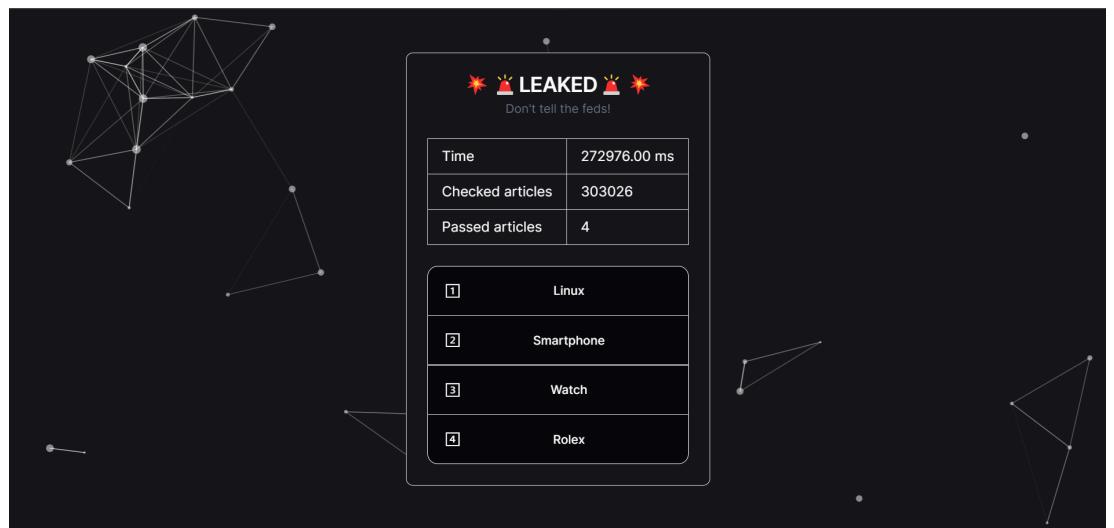
## B. Pengujian 2

Judul Awal : Linux

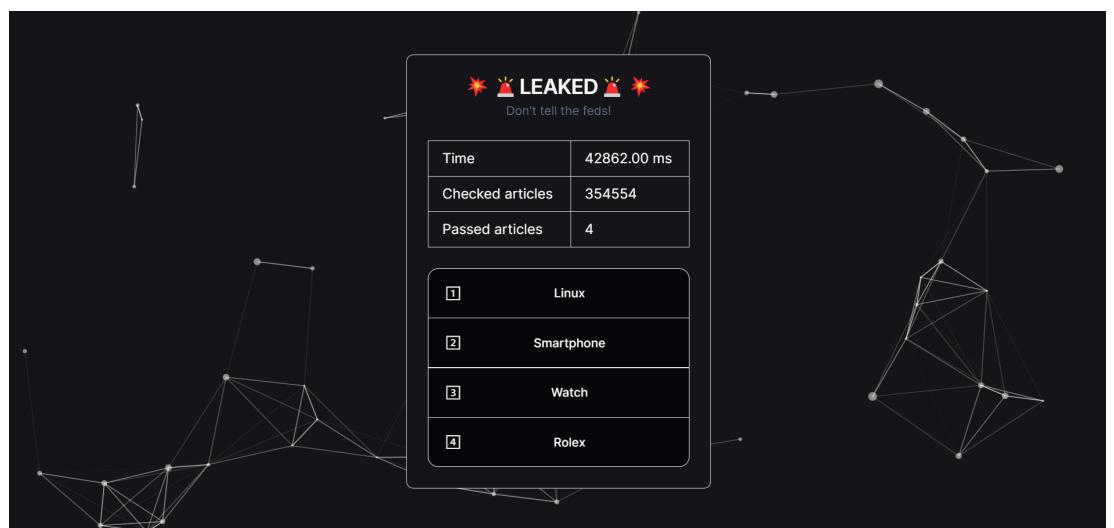
Judul Tujuan : Rolex

Algoritma : BFS

Solusi : Pertama



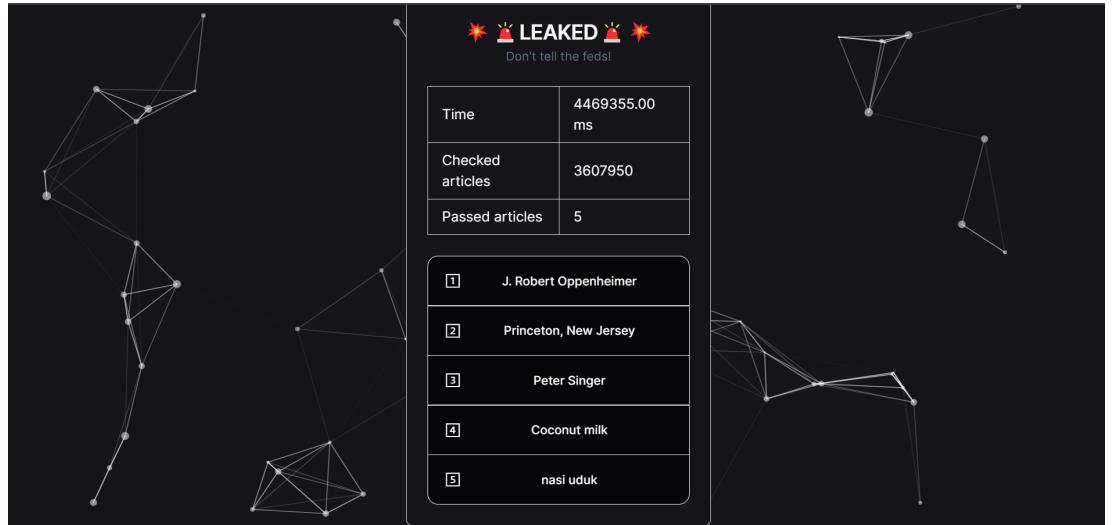
Gambar 1. BFS dengan Satu Solusi



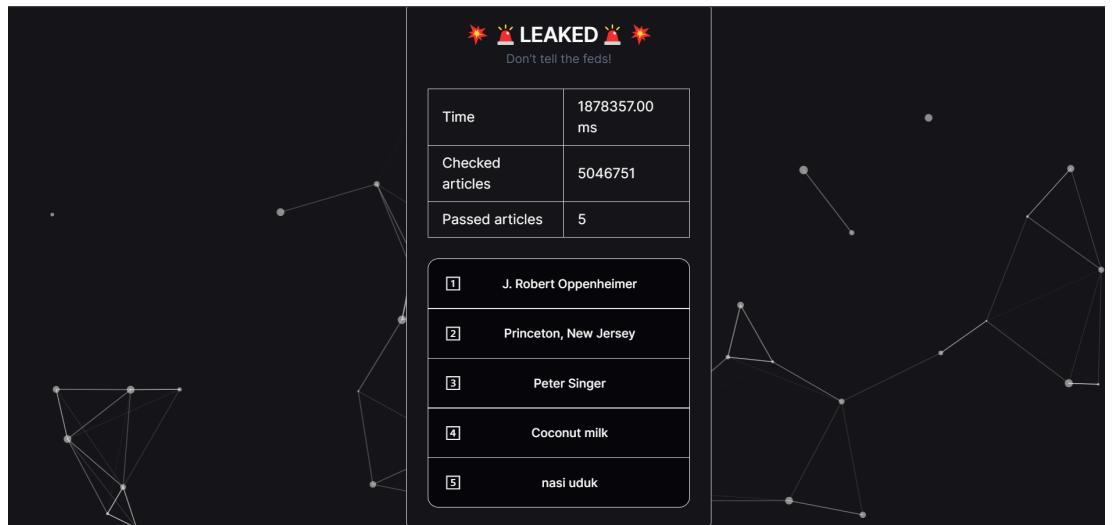
Gambar 2. IDS dengan Satu Solusi

### C. Pengujian Ketiga

- Judul Awal : J. Robert Oppenheimer  
 Judul Tujuan : Nasi Uduk  
 Algoritma : BFS  
 Solusi : Pertama

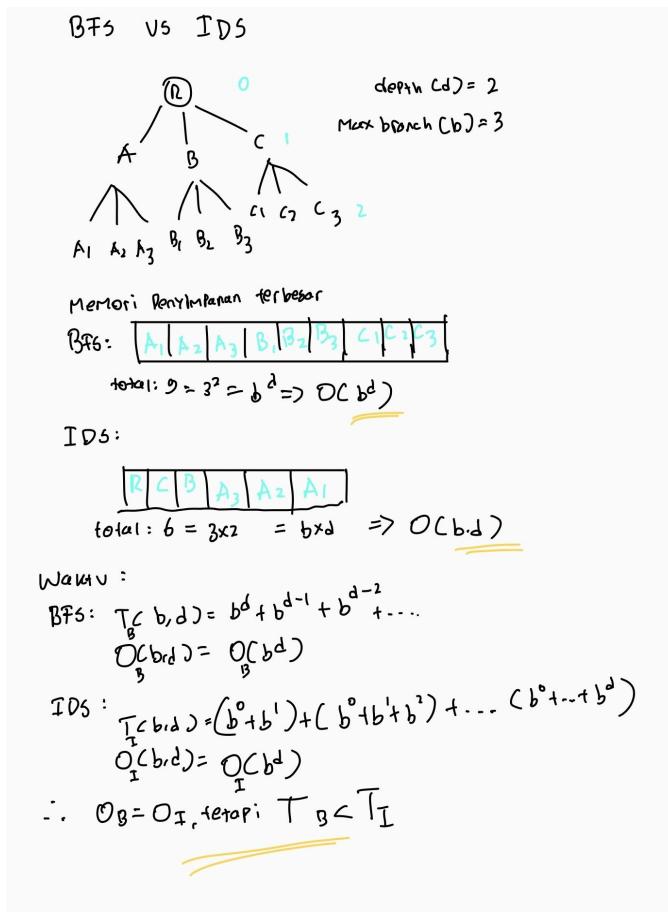


Gambar 1. BFS dengan Satu Solusi



Gambar 2. IDS dengan Satu Solusi

## 4.4 Analisis Hasil Pengujian



Gambar 1. BFS vs IDS

dengan BFS yakni  $O(b^d)$ , tetapi  $T(b,d)$  dari BFS  $<$  IDS karena IDS merupakan algoritma yang mengiterasi secara iteratif dari depth 1 hingga  $d$ . Dari hal tersebut, tampak bahwa BFS merupakan algoritma yang lebih unggul dibandingkan IDS. Akan tetapi, perlu dipertimbangkan kembali terkait harga memori (*memory cost*) dari kedua algoritma tersebut. Jika kita melihat dari kompleksitas memori (*memory complexity*), maka algoritma IDS jauh lebih unggul karena hanya menyimpan informasi branch yang sedang ditelusuri dibandingkan BFS. Untuk lebih jelaskan, dari gambar 1 didapatkan bahwa algoritma IDS memiliki kompleksitas memori yang lebih efisien yakni  $O(b \cdot d)$  dibandingkan BFS yakni  $O(b^d)$ . Oleh karena itu, kedua algoritma memiliki keunggulan dan kekurangannya masing-masing. Jika kita mementingkan waktu yang cepat dalam melakukan permainan *wiki race* maka algoritma BFS merupakan solusi yang tepat. Akan tetapi, jika kita mementingkan efisiensi dalam penggunaan memori akan tetapi ingin waktu dijalankan juga tetap memadai, maka algoritma IDS merupakan solusi yang tepat.

Berdasarkan hasil pengujian didapatkan bahwa waktu hasil pencarian BFS lebih cepat dari IDS. Meskipun dalam beberapa momen, IDS terkadang lebih cepat dari BFS, hal tersebut disebabkan oleh koneksi atau jaringan internet yang tidak stabil dalam proses pencarian. Selain internet, spesifikasi perangkat juga memengaruhi seperti jumlah RAM yang dimiliki dan jumlah *logical processor* yang dapat dipakai dalam *multiprocessing*. Jika kita mengesampingkan itu dan melihat berdasarkan teori, waktu pencarian BFS lebih cepat dibandingkan IDS seperti yang telah dibuktikan dari gambar 1. Dari gambar 1, didapatkan bahwa meskipun kompleksitas waktu Big O dari IDS sama

## **BAB V**

## **PENUTUP**

### **5.1 Kesimpulan**

Pencarian *wiki race* tidak hanya bergantung pada algoritma nya, tetapi juga faktor-faktor eksternal seperti keadaan koneksi atau jaringan internet, spesifikasi RAM, dan spesifikasi CPU. Di samping itu, jika kita asumsi bahwa faktor-faktor eksternal hilang maka algoritma BFS merupakan algoritma yang lebih cepat dalam mencari link dibandingkan dengan IDS. Akan tetapi, algoritma IDS memiliki kualitas penggunaan memori yang lebih hemat dari BFS. Oleh karena itu, pemilihan algoritma dalam permainan *wiki race* dapat disimpulkan bahwa menggunakan algoritma BFS jika mementingkan efektivitas waktu dan menggunakan algoritma IDS jika mementingkan efisiensi memori.

### **5.2 Saran**

Saran bagi penyelenggara tugas besar ini adalah lebih spesifik dalam menetapkan ketentuan tugasnya dari awal, karena terdapat beberapa perubahan yang cukup major yang mempengaruhi penggeraan mahasiswa.

### **5.3 Refleksi**

Tugas besar ini memberikan kami kesempatan untuk mengaplikasikan algoritma IDS dan BFS yang biasanya kami pelajari secara teori saja. Kami juga dikenalkan dengan bahasa Golang yang tentu saja menarik karena memiliki *style coding* yang cukup berbeda dari bahasa pemrograman lainnya.

## **LAMPIRAN**

### **6.1 GitHub Repository (Latest Release)**

[https://github.com/owenthe10x/Tubes1\\_Queen](https://github.com/owenthe10x/Tubes1_Queen)

## **REFERENSI**

Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1).

Diakses pada 25 Februari 2024, dari

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf>

Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 2).

Diakses pada 25 Februari 2024, dari

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf>