

LAPORAN TUGAS BESAR 2  
IF2211 Strategi Algoritma  
PEMANFAATAN ALGORITMA IDS DAN BFS  
DALAM PERMAINAN WIKI RACE



Disusun Oleh :  
Muhammad Gilang Ramadhan (13520137)  
Kharris Khisunica (13522051)

PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
2024

# Daftar Isi

<b>Daftar Isi</b>	<b>2</b>
<b>1. Deskripsi Tugas</b>	<b>3</b>
1.1. Deskripsi Tugas Besar	3
1.2. Spesifikasi Tugas Besar	3
<b>2. Landasan Teori</b>	<b>3</b>
2.1 Teori Graf	3
2.2 Algoritma IDS	4
2.3 Algoritma BFS	4
2.4 Jenis-jenis artikel dalam wikipedia	4
<b>3. Analisis Pemecahan Masalah</b>	<b>4</b>
3.1 BFS (Breadth First Search)	4
3.2 Iterative Deepening Search (IDS)	7
3.3 Fitur Fungsional Website	9
<b>4. Implementasi dan Pengujian</b>	<b>10</b>
4.1 Tata Cara Penggunaan Website	10
4.2 Tes Kasus	10
4.3 Kode Program	12
<b>5. Kesimpulan dan Saran</b>	<b>18</b>
5.1 Kesimpulan	18
5.2 Saran	19
<b>LAMPIRAN</b>	<b>20</b>
<b>DAFTAR PUSTAKA</b>	<b>20</b>

# 1. Deskripsi Tugas

## 1.1. Deskripsi Tugas Besar

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 Besar

- Program dibuat dalam bahasa Go dan 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.
- Program wajib dapat mencari rute terpendek kurang dari 5 menit untuk setiap permainan.

# 2. Landasan Teori

## 2.1 Teori Graf

Graf adalah pasangan terurut dari pasangan  $G = (V, E)$  yang terdiri dari himpunan tidak kosong  $V$  yang berisi simpul  $\{v_1, v_2, \dots, v_n\}$ , dan himpunan  $E$  yang berisi sisi yang menghubungkan sepasang simpul  $\{e_1, e_2, \dots, e_m\}$ . Hubungan antara simpul dan sisi bisa dinotasikan sebagai  $e_p = (v_i, v_j)$ , dimana  $v_i, v_j \in V, e_p \in E$ .

## 2.2 Algoritma IDS

Algoritma *Iterative-Depth Search* (IDS) adalah salah satu algoritma untuk mencari data yang diinginkan dalam sebuah struktur data graf. Algoritma ini memakai pendekatan *Depth-First Search* (DFS) yang terbatas pada kedalaman tertentu. *Depth-First Search* (DFS) adalah algoritma yang mengunjungi setiap simpul secara transversal.

## 2.3 Algoritma BFS

Algoritma *Breadth-First Search* (BFS) adalah salah satu algoritma untuk mencari data yang diinginkan dalam sebuah struktur data graf. Pencarian dimulai dari simpul akar lalu data dicari secara melebar dengan mengunjungi semua simpul yang bertetangga dengan simpul tersebut terlebih dahulu. Setelah itu, kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang telah dikunjungi.

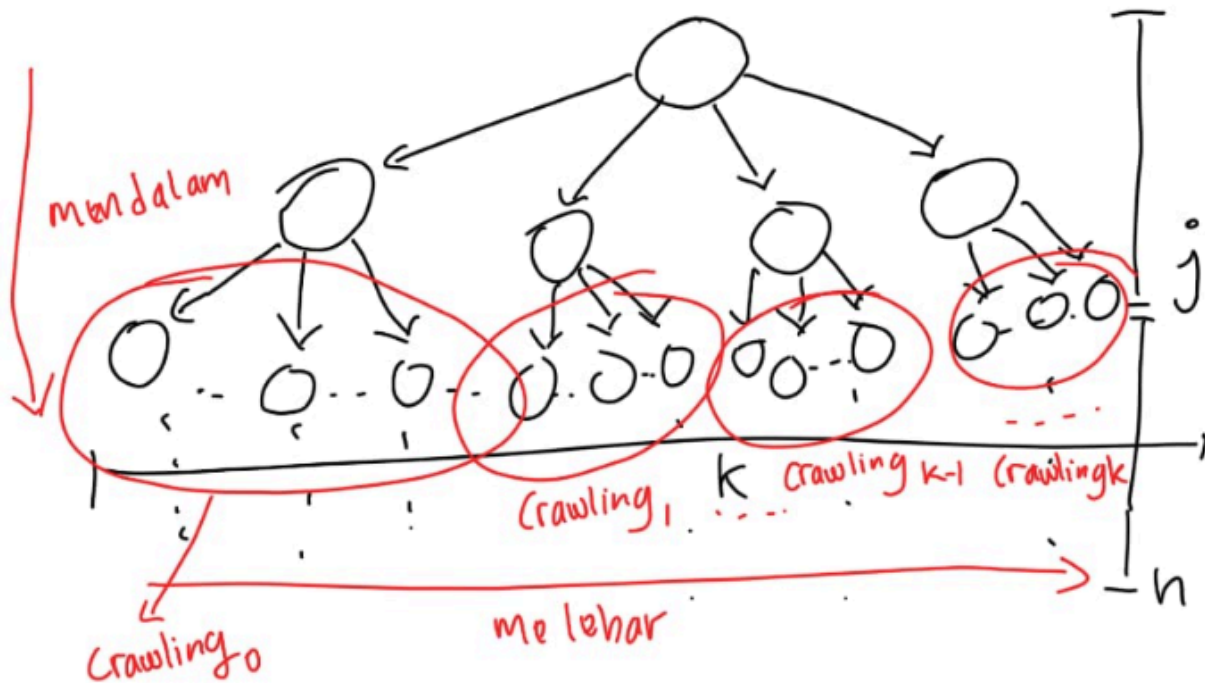
## 2.4 Jenis-jenis artikel dalam wikipedia

Di dalam wikipedia, terdapat beberapa jenis artikel spesial yang bisa mempengaruhi hasil dari permainan WikiRace. Jenis pertama adalah artikel yang tidak bisa diakses oleh artikel lain atau artikel *Orphan*/Yatim piatu. Per tanggal 27 April 2024, terdapat 60 ribu lebih artikel yang termasuk *Orphan*. Jenis kedua adalah artikel yang tidak bisa mengakses artikel lain atau artikel *Dead end*/Jalan buntu. Per tanggal 27 April 2024, hanya terdapat 129 artikel yang termasuk *Dead end*.

# 3. Analisis Pemecahan Masalah

## 3.1 BFS (Breadth First Search)

Adapun algoritma BFS basic yang diimplementasikan pada permasalahan ini ialah dengan melakukan penelusuran secara melebar dimana untuk setiap level dilakukan pembangkitan anak dari masing-masing node yaitu pada saat proses crawling pada scrapping data dari wikipedia. Adapun setiap scrapping tersebut akan dihasilkan depth baru yaitu  $\text{current depth} + 1$ . Adapun pembangkitan tersebut juga dilakukan setelah proses iterasi pada semua node pada satu level itu telah selesai. Adapun proses tersebut dapat dilihat melalui ilustrasi graph berikut.



#### Sumber Dokumentasi Pribadi Penulis

Pada proses pembangkitan tersebut, kelompok kami menggunakan konsep caching dan multithreading terhadap setiap proses scrapping data pada saat mendapatkan node baru tersebut. Adapun pada proses multithreading tersebut, proses pembangkitan node dilakukan dengan menggunakan bantuan node worker yang dialokasikan secara asynchronus dengan menggunakan Core CPU pengguna, yang mana logic tersebut dapat dilihat melalui kode berikut.

```
import (
    "log"
    "runtime"
    "sync"
)

var numNodesPerLevelBFS int = runtime.NumCPU() * 10
```

Proses alokasi jumlah node terhadap jumlah CPU yang akan runtime

```
// Initialize the output channel and wait group
outputCh := make(chan ArticleInfo)
// Wait group to wait for all the goroutines to finish
var wg sync.WaitGroup
wg.Add(1)
// Start the first goroutine to scrap the start article
go scrappedArticleAndSync(startArticle, outputCh, &wg)
go closeChannelOnWg(outputCh, &wg)
```

**Proses single scrapping secara asynchronous dengan node worker**

```
// Start the next level
inputCh := make(chan string)
nextOutputCh := make(chan ArticleInfo, 1000)
var nextWg sync.WaitGroup
nextWg.Add(numNodesPerLevelBFS)
for i := 0; i < numNodesPerLevelBFS; i++ {
    go scrappedArticlesAndSync(inputCh, nextOutputCh, &nextWg)
}
```

**Proses scrapping pada multi-article secara asynchronous dengan node worker**

Adapun kelompok kami juga menerapkan sistem caching untuk menyimpan state scrapping yang telah dilakukan sebelumnya pada memory sementara server. Oleh karena itu, pada saat dilakukan scrapping pada URL yang sama, maka backend tidak perlu melakukan scrapping ulang, melainkan cukup mengambil data tersebut yang disimpan secara mapping pada cache memory.

Sementara itu, penelusuran berakhir ketika state sudah mencapai goal node ataupun sudah mencapai batas maksimum dari komputasi dari PC pengguna, karena jika infinity juga bakal tetap dibatasi oleh timeout otomatis dari server.

Dari segi kompleksitas, jelas bahwa untuk kompleksitas waktu ditentukan oleh seberapa banyak node yang dibangkitkan dan traversal. Sehingga kompleksitas waktunya ialah  $O(V+E)$  dengan E merupakan kompleksitas node dan E merupakan kompleksitas edge yang untuk banyaknya edge itu sama dengan banyaknya child pada suatu node yang diperoleh dari proses scrapping data hyperlink. Sementara itu, untuk kompleksitas ruang ditentukan oleh banyaknya node pada suatu level saja. Karena sejatinya jika menyimpan semua node dari awal sampai habis tentu akan membebankan kepada server, oleh karena itu untuk setiap level node yang sebelum-sebelumnya itu dihapus, jika ingin dilakukan tracking pada path solusi kita dapat menyimpan node-node yang dibangkitkan secara unik, jadi untuk 2 node yang sama cukup disimpan 1 saja kemudian dapat menggunakan informasi parent dan child untuk melakukan iterasi pada setiap path.

Kemudian, ketika ada 2 node yang sama untuk node 1 yang level nya  $<$  level node 2, maka kelompok kami menggunakan optimasi untuk tidak memasukkan node tersebut sebagai next node untuk ditraversal

maupun dikunjungi selanjutnya. Karena node yang level lebih tinggi tersebut namun mempunyai value nama artikel yang sama, sudah pasti tidak shortest path, karena untuk mendapatkan next node yang sama haruslah depthnya akan lebih besar daripada node yang sebelumnya itu.

### 3.2 IDS (Iterative Deepening Search)

Secara umum konsep iterative deepening search yang diimplementasikan kelompok kami adalah sama dengan konsep IDS pada umumnya, yaitu dengan menggunakan konsep mirip dengan DFS namun ditambahkan batas maksimum penelusuran level pada setiap iterasi pada kedalaman tertentu. Jadi, dengan melalui pendekatan ini, traversal tidak mungkin menjadi infinity (tak hingga). Bisa dimanfaatkan kedalaman yang dilakukan untuk melakukan optimasi terhadap penelusuran graph yang dilakukan. Yang dilakukan secara mendalam kemudian backtracking ke level terakhir pada node yang berada bersebelahan dengan node yang telah ditelusuri secara maksimum pada node sekarang.

Adapun pada kelompok kami, seperti halnya pada BFS, dilakukan optimasi proses scrapping dengan menggunakan caching yang disimpan secara mapping pada memory backend. Kemudian, juga dilakukan runtime secara asynchronous dengan memanfaatkan komputasi asynchronous pada goroutine pada node worker pada proses mendapatkan child untuk setiap ekspansi ke kedalaman baru. Untuk setiap node yang dicapai, aplikasi ini akan memeriksa apakah node tersebut merupakan node target atau bukan. Apabila iya, aplikasi akan menambahkan jalur yang ditemukan ke dalam daftar jalur unik.

Adapun digunakan beberapa goroutine untuk mengolah URL dari setiap node. Setiap goroutine akan mengambil URL dari antrian URL, mengakses halaman Wikipedia yang relevan, mengekstrak semua hyperlink dari halaman tersebut, dan menambahkannya ke dalam antrian hasil. Jika terjadi kesalahan selama proses ini, goroutine akan menambahkan kesalahan tersebut ke dalam antrian kesalahan. Setelah semua node telah dicapai atau node target telah ditemukan, aplikasi ini akan menutup semua antrian dan menunggu sampai semua goroutine selesai. Setelah itu, aplikasi akan mengembalikan hasil akhir, jumlah artikel yang diperiksa, jumlah artikel yang dicapai, jumlah jalur yang ditemukan, dan waktu yang diperlukan untuk menyelesaikan pencarian

Yang mana hal tersebut akan menambah kecepatan dari segi komputasi wikipedia race yang diimplementasikan oleh kelompok kami. Adapun proses tersebut dapat dilihat pada potongan kode berikut.

```

func closeChannelOnWg(ch chan ArticleInfo, wg *sync.WaitGroup) {
    (*wg).Wait()
    close(ch)
}

func feedArticlesIntoChannel(articles []string, ch chan string) {
    for _, article := range articles {
        ch <- article
    }
    close(ch)
}

func threadScrappingProcessing(resultQueue chan<- []Node, errQueue chan<- error, urlQueue <-chan string, wg *
    for url := range urlQueue {
        ch := make(chan []Node)
        errCh := make(chan error)

        go getChilds(url, ch, errCh)

        select {
        case result := <-ch:
            resultQueue <- result
        case err := <-errCh:
            errQueue <- err
        }
    }

    wg.Done()
}

func multithreadScrappingProcessing(resultQueue chan<- []Node, errQueue chan<- error, urlQueue <-chan string,
    for i := 0; i < NumOfNodeWORKERS; i++ {
        wg.Add(1)
        go threadScrappingProcessing(resultQueue, errQueue, urlQueue, wg)
    }
}

```

### Potongan kode untuk proses scraping dengan queue pada node worker

Untuk kompleksitas yang diperoleh juga tergantung dengan maximum depth yang menjadi parameter pada IDS, yang mana semakin besar maximum depth maka iterasi akan dilakukan lebih dalam, oleh karena itu node-node yang dibangkitkan pada proses ekspansi level akan mempengaruhi banyaknya node yang akan dikunjungi selanjutnya. Untuk kompleksitas ruangnya juga tergantung dengan seberapa dalam untuk satu bagian path dari atas kebawah yang akan dilakukan backtracking (menghapus node-node yang levelnya dibawah dari node backtracking tersebut).



### 3.3 Fitur Fungsional Website

#### Graph Visualization

Choose Algorithm:

IDS

Source Title:

Dracula

Goal Title:

Sweden

Max Depth:

4

Find All: ☒

Generate Graph

#### Tampilan Menu Input Website

Website menyediakan menu inputan untuk memilih jenis algoritma yang akan digunakan (IDS atau BFS), judul artikel awal, judul artikel tujuan, dan kedalaman maksimum untuk IDS. Selain itu, terdapat opsi untuk menampilkan semua solusi.

## Graph Visualization

### Response Data

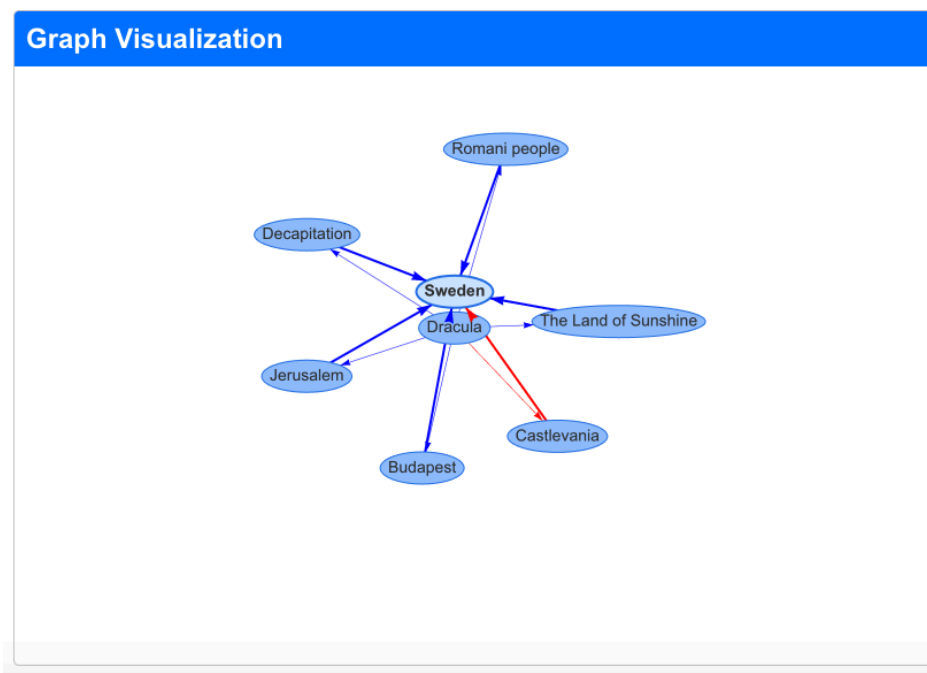
Response Message: success

Elapsed Time: 27.468888792 Seconds

Number of Articles Checked: 416

Number of Node Articles Visited: 153498

Number of Paths: 6



## Tampilan Output Website

Website menampilkan hasil dari masukkan user. Diberikan informasi apakah artikel berhasil ditemukan, waktu yang diperlukan untuk mencari jalur antar artikel, banyak artikel yang diperiksa, banyak artikel yang dilewati, dan banyak rute yang ditemukan antar kedua artikel tersebut.

## 4. Implementasi dan Pengujian

### 4.1 Tata Cara Penggunaan Website

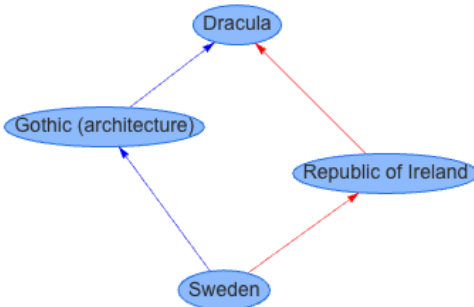
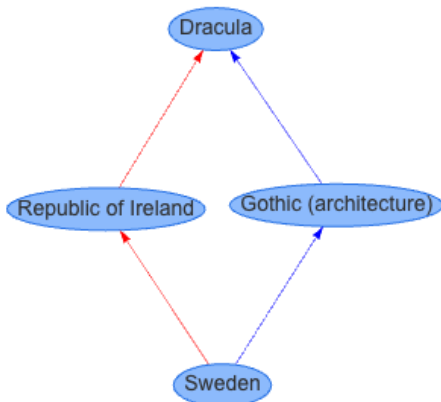
- User memilih jenis algoritma yang ingin digunakan, apakah ingin menggunakan algoritma IDS atau BFS.
- User kemudian memasukkan judul artikel awal dan judul artikel akhir yang sudah dipastikan ada di Wikipedia.
- Jika user memilih algoritma IDS, maka user juga diminta untuk memasukkan kedalaman maksimum.
- User juga diberi pilihan apakah ingin menampilkan seluruh rute yang merupakan solusi dengan check box. Jika user tidak mencentangnya maka website hanya akan menampilkan salah satu solusi saja.

### 4.2 Tes Kasus

#### 1. Kasus 1

*Tabel 4.3.2 Tabel Tes Kasus 1*

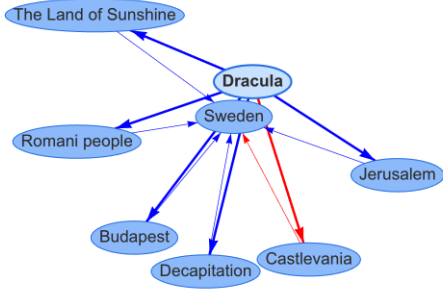
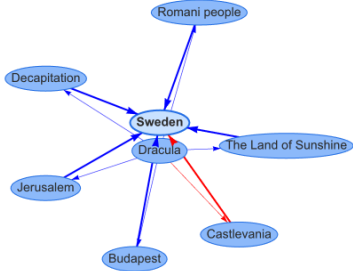
Artikel Awal		Sweden
Artikel Akhir		Dracula
Jumlah artikel yang diperiksa	IDS	1291
	BFS	1456
Jumlah artikel yang dikunjungi	IDS	736886
	BFS	8021235

Jalur yang dilalui	IDS	
	BFS	
Jumlah waktu yang diperlukan	IDS	273.502542 ms
	BFS	275. 73124 ms

## 2. Kasus 2

Tabel 4.3.2 Tabel Tes Kasus 2

Artikel Awal		Dracula
Artikel Akhir		Sweden
Jumlah artikel yang diperiksa	IDS	5166
	BFS	41628
Jumlah artikel yang dikunjungi	IDS	153498
	BFS	124623

Jalur yang dilalui	IDS	
	BFS	
Jumlah waktu yang diperlukan	IDS	27468.88792 ms
	BFS	25621.9424 ms

## 4.3 Kode Program

Berikut dilampirkan kode program utama untuk algoritma BFS dan IDS yang diimplementasikan.

### Algoritma Utama IDS

```
package utils
```

```
import (
```

```
"fmt"
```

```
"log"
```

```
"strings"
```

```
"sync"
```

```

)

func isSearchCondition(flag bool, depth int, foundCount int) bool {
if !flag {
return foundCount == 0
}

return depth <= foundCount
}

// IDS Algorithm
func getShortestPathIDS(start Node, target Node, maxDepth int, findAll bool) (int, int, int, [][]string) {
var numOfArticlesChecked int
var articlesLoop int
var foundCount int = maxDepth + 1

if !findAll {
foundCount = 0
}

urlQueue := make(chan string, 500)
resultQueue := make(chan []Node, 500)
errQueue := make(chan error, 500)
results := make([][]string, 0)
pathSet := make(map[string]bool)

// create multithread to get the scrapping data
var wg sync.WaitGroup
multithreadScrappingProcessing(resultQueue, errQueue, urlQueue, &wg)

// iterate through increasing depth
for depth := 0; depth <= maxDepth && isSearchCondition(findAll, depth, foundCount); depth++ {
visited := make(map[string]bool)
var stack []Node

// set the initial path for the start node
start.Path = []string{start.Title}
stack = append(stack, start)

// iterate through the stack
for len(stack) > 0 && (foundCount == 0 || findAll) {
// pop the last element from the stack

```

```

current := stack[len(stack)-1]
pathKey := strings.Join(current.Path, "^")
stack = stack[:len(stack)-1]

// check if the current node is the target
if current.Title == target.Title && !pathSet[pathKey] {
results = append(results, current.Path)
pathSet[pathKey] = true
if findAll && depth < foundCount {
foundCount = depth
} else if !findAll {
foundCount = 1
break
}
}

if len(current.Path) > depth || visited[current.URL] {
continue
}

// mark the current node as visited
visited[current.URL] = true
urlQueue <- current.URL

// get the neighbors of the current node
select {
case neighbors, ok := <-resultQueue:
if !ok {
// resultQueue has been closed
break
}
for _, neighbor := range neighbors {
if !visited[neighbor.URL] {
neighbor.Path = append([]string(nil), current.Path...)
neighbor.Path = append(neighbor.Path, neighbor.Title)
stack = append(stack, neighbor)
articlesLoop++
}
}
numOfArticlesChecked++
case err := <-errQueue:
log.Println(err)
// Optionally handle error and continue the loop

```

```

}
}
}

close(urlQueue)
wg.Wait()
close(resultQueue)
close(errQueue)

numberOfPath := len(results)

if numberOfPath != 0 {
fmt.Printf("Successfully get the Result: %v\n", results)
} else {
fmt.Println("No path found.")
}

return numOfArticlesChecked, articlesLoop, numberOfPath, results
}

```

## Algoritma Utama BFS

```

package utils

import (
    "log"
    "runtime"
    "sync"
)

var numNodesPerLevelBFS int = runtime.NumCPU() * 10

func getPathBFS(articleToParent *map[string]string, endArticle string) *[]string {
    // Initialize an empty slice to store the reversed path
    reversedPath := make([]string, 0)

    // Start from the end article and trace back to the root
    currentArticle := endArticle
    for currentArticle != "root" {
        // Add the current article to the reversed path
        reversedPath = append(reversedPath, currentArticle)
    }
}

```

```

// Move to the parent article
currentArticle = (*articleToParent)[currentArticle]
}

// Calculate the length of the reversed path
pathLen := len(reversedPath)

// Initialize a new slice to store the final path
path := make([]string, pathLen)

// Reverse the order of articles to get the correct path
for i := 0; i < pathLen; i++ {
// Get the URL string from the reversed path and store it in the correct order
path[i] = URL_SCRAPPING_WIKIPEDIA + reversedPath[pathLen-i-1]
}

// Return a pointer to the final path slice
return &path
}

func GetShortestPathBFS(startUrl string, endUrl string) (*[]string, *map[string]string) {
// Initialize the map to store the parent of each article
articleToParent := make(map[string]string)
emptyPath := make([]string, 0)

// start url and end url should be reachable
startArticle, err := GetArticleNameFromURLString(startUrl)

// if the start url is not reachable, return empty path
if err != nil || !IsReachable(startUrl) {
log.Printf("Invalid StartURL: %s\n", startUrl)
return &emptyPath, &articleToParent
}

// start url and end url should be reachable
endArticle, err := GetArticleNameFromURLString(endUrl)

// if the end url is not reachable, return empty path
if err != nil || !IsReachable(endUrl) {
log.Printf("Invalid EndURL: %s\n", endUrl)
return &emptyPath, &articleToParent
}

// Initialize the parent of the start article as "root"

```



```

articleToParent[startArticle] = "root"

// If the start and end articles are the same, return the path with the start article
if startUrl == endUrl {
    emptyPath = append(emptyPath, startUrl)
    return &emptyPath, &articleToParent
}

// Initialize the output channel and wait group
outputCh := make(chan ArticleInfo)
// Wait group to wait for all the goroutines to finish
var wg sync.WaitGroup
wg.Add(1)
// Start the first goroutine to scrap the start article
go scrappedArticleAndSync(startArticle, outputCh, &wg)
go closeChannelOnWg(outputCh, &wg)

level := 0
found := false

for {
    // Collect outputs from scrapping at the current level
    // Skip articles that are already visited
    scrappedDatas := make([]string, 0)
    for articleWithParent := range outputCh {
        nextArticle := articleWithParent.Article
        if articleToParent[nextArticle] == "" {
            articleToParent[nextArticle] = articleWithParent.ParentArticle
            if nextArticle == endArticle {
                found = true
                break
            }
            scrappedDatas = append(scrappedDatas, nextArticle)
        }
    }

    log.Printf("Level %d\n", level)
    // If the solution is found, break the loop
    if found {
        log.Printf("Successfully Found the Solution!")
        break
    }
}

```

```

// If no more articles to scrap, break the loop
log.Printf(
    "Collected %d outputs from scrapping data the child\n",
    len(scrapedDatas))
level++

// Start the next level
inputCh := make(chan string)
nextOutputCh := make(chan ArticleInfo, 1000)
var nextWg sync.WaitGroup
nextWg.Add(numNodesPerLevelBFS)
for i := 0; i < numNodesPerLevelBFS; i++ {
    go scrappedArticlesAndSync(inputCh, nextOutputCh, &nextWg)
}

// close the output channel when all the goroutines are done
go closeChannelOnWg(nextOutputCh, &nextWg)
log.Printf("Level %d: Started %d Scrapeds\n", level, numNodesPerLevelBFS)

// feed articles into the input channel
go feedArticlesIntoChannel(scrapedDatas, inputCh)
outputCh = nextOutputCh
}

return getPathBFS(&articleToParent, endArticle), &articleToParent
}

```

## 5. Kesimpulan dan Saran

### 5.1 Kesimpulan

Dari tes kasus yang ada, didapatkan bahwa algoritma IDS cenderung lebih lambat dari algoritma BFS. Hal ini dikarenakan dalam tugas ini, graf yang dihasilkan memiliki branching factor yang cenderung cukup besar jika dibandingkan dengan kedalamannya. Algoritma-algoritma tersebut diimplementasikan menggunakan bahasa pemrograman GO tersebut mendukung proses komputasi yang dilakukan menjadi lebih cepat. Juga menggunakan kakas lain seperti React pada frontend membantu proses visualisasi menjadi lebih interaktif. Program kami juga dapat menemukan berbagai solusi rute terpendek dari satu artikel ke artikel lainnya, tergantung pada kedalaman pencarian (depth), di mana semakin dalam depth,

semakin banyak kemungkinan solusi yang ditemukan karena terjadi proses pembangkitan node-node baru yang melibatkan proses scrapping data.

## 5.2 Saran

Saran untuk pembuatan proyek serupa berikutnya adalah mencoba menggunakan algoritma dan kakas lain yang bisa mempercepat dan mengefisienkan proses pencarian sehingga bisa mengurangi waktu yang diperlukan.

# LAMPIRAN

Link Github: [https://github.com/gilangr301102/Tubes2\\_Gass](https://github.com/gilangr301102/Tubes2_Gass)

# DAFTAR PUSTAKA

<https://informatika.stei.itb.ac.id/~rinaldi.munir>

<https://www.zenrows.com/blog/goquery#build-initial-golang-scrapers>

<https://go.dev/doc/>

<https://www.youtube.com/watch?v=JheGL6uSF-4>