

LAPORAN TUGAS BESAR 2

IF2211 STRATEGI ALGORITMA

Pemanfaatan Algoritma IDS dan BFS dalam Permainan WikiRace



Disusun oleh:

Agil Fadillah Sabri 13522006

Raden Rafly Hanggaraksa B 13522014

Daniel Mulia Putra Manurung 13522043

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2024

DAFTAR ISI

| | |
|--|-----------|
| DAFTAR ISI..... | 1 |
| DAFTAR GAMBAR..... | 2 |
| DAFTAR TABEL..... | 3 |
| BAB I DESKRIPSI MASALAH..... | 4 |
| BAB II LANDASAN TEORI..... | 5 |
| 2.1. Dasar Teori..... | 5 |
| 2.2. Penjelasan Web yang Dibangun..... | 6 |
| BAB III ANALISIS PEMECAHAN MASALAH..... | 7 |
| 3.1. Langkah-Langkah Pemecahan Masalah..... | 7 |
| 3.2. Pemetaan Masalah..... | 9 |
| 3.3. Fitur dan Arsitektur Website..... | 9 |
| 3.4. Ilustrasi Kasus..... | 12 |
| BAB IV IMPLEMENTASI DAN PENGUJIAN..... | 14 |
| 4.1. Spesifikasi Program..... | 14 |
| 4.2. Cara Penggunaan Program..... | 28 |
| 4.3. Hasil Pengujian..... | 30 |
| 4.4. Analisis Hasil Pengujian..... | 39 |
| BAB V PENUTUP..... | 46 |
| 5.1. Kesimpulan..... | 46 |
| 5.2. Saran..... | 46 |
| 5.3. Refleksi..... | 47 |
| LAMPIRAN..... | 48 |
| DAFTAR PUSTAKA..... | 49 |

DAFTAR GAMBAR

| | |
|---|----|
| Gambar 1. Ilustrasi Graf WikiRace..... | 4 |
| Gambar 2. Graf Solusi..... | 9 |
| Gambar 3. Input Checking..... | 10 |
| Gambar 4. Hasil Pencarian Solusi Banyak..... | 10 |
| Gambar 5. Hasil Pencarian Solusi Tunggal..... | 11 |
| Gambar 6. Tombol Pemilihan Algoritma Pencarian..... | 11 |
| Gambar 7. Saran Judul Pencarian..... | 12 |
| Gambar 8. Koleksi Tombol..... | 13 |
| Gambar 9. Ilustrasi Tampilan Hasil Pencarian..... | 13 |
| Gambar 10. Laman Utama Program..... | 29 |
| Gambar 11. Output Program..... | 29 |
| Gambar 12. Hasil Test Case 1 BFS Satu Solusi..... | 30 |
| Gambar 13. Hasil Test Case 1 BFS Banyak Solusi..... | 30 |
| Gambar 14. Hasil Test Case 2 BFS Satu Solusi..... | 31 |
| Gambar 15. Hasil Test Case 2 BFS Banyak Solusi..... | 31 |
| Gambar 16. Hasil Test Case 3 BFS Satu Solusi..... | 32 |
| Gambar 17. Hasil Test Case 3 BFS Banyak Solusi..... | 32 |
| Gambar 18. Hasil Test Case 1 IDS Satu Solusi tanpa Cache..... | 33 |
| Gambar 19. Hasil Test Case 1 IDS Satu Solusi dengan Cache..... | 33 |
| Gambar 20. Hasil Test Case 1 IDS Banyak Solusi tanpa Cache..... | 34 |
| Gambar 21. Hasil Test Case 1 IDS Banyak Solusi dengan Cache..... | 34 |
| Gambar 22. Hasil Test Case 2 IDS Satu Solusi tanpa Cache..... | 35 |
| Gambar 23. Hasil Test Case 2 IDS Satu Solusi dengan Cache..... | 35 |
| Gambar 24. Hasil Test Case 2 IDS Banyak Solusi tanpa Cache..... | 36 |
| Gambar 25. Hasil Test Case 2 IDS Banyak Solusi dengan Cache..... | 36 |
| Gambar 26. Hasil Test Case 3 IDS Satu Solusi tanpa Cache..... | 37 |
| Gambar 27. Hasil Test Case 3 IDS Satu Solusi dengan Cache..... | 37 |
| Gambar 28. Hasil Test Case 3 IDS Banyak Solusi tanpa Cache..... | 38 |
| Gambar 29. Hasil Test Case 3 IDS Banyak Solusi dengan Cache..... | 38 |
| Gambar 30. Grafik Execution Time IDS..... | 43 |

DAFTAR TABEL

| | |
|--|----|
| Tabel 1. Tabel Hasil Uji Coba Algoritma BFS..... | 32 |
| Tabel 2. Tabel Hasil Uji Coba Algoritma IDS..... | 38 |
| Tabel 3. Waktu Eksekusi BFS dalam detik..... | 39 |
| Tabel 4. Banyak Artikel yang Dikunjungi Selama Pencarian BFS..... | 40 |
| Tabel 5. Waktu Eksekusi IDS dalam detik..... | 42 |
| Tabel 6. Banyak Artikel yang Dikunjungi Selama Pencarian BFS..... | 43 |

BAB I

DESKRIPSI MASALAH

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 mengklik 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

BAB II

LANDASAN TEORI

2.1. Dasar Teori

2.1.1. Penjelajahan Graf

Penjelajahan Graf adalah sebuah proses pengunjungan setiap simpul dalam sebuah graf untuk mendapatkan hasil tertentu. Dalam ilmu komputer, penjelajahan graf memiliki implementasi yang sangat banyak, mulai dari pencarian nilai maksimum, pencarian rute paling optimal, dan banyak contoh kasus lainnya. Penjelajahan graf memungkinkan terjadinya banyak redundansi, dikarenakan tidak terdapat jaminan di mana sebuah simpul yang akan kita kunjungi pernah berhubungan dengan simpul lainnya yang telah kita kunjungi. Beberapa algoritma penjelajahan graf yang sering digunakan adalah *Depth First Search*, *Breadth First Search*, *Depth Limited Search* dan *Iterative Deepening Search*.

2.1.2. Breadth First Search (BFS)

Breadth First Search adalah salah satu metode atau algoritma penjelajahan dalam graf yang mencari seluruh anak simpul dari sebuah simpul yang telah dikunjungi sebelumnya. Algoritma ini biasanya bekerja dengan menggunakan sebuah *queue*. Untuk setiap simpul yang dikunjungi, algoritma ini akan menambahkan anak simpul tersebut dalam sebuah *queue*, lalu akan melakukan pencarian kembali sesuai dengan urutan simpul yang terdapat dalam *queue* tersebut. Kompleksitas waktu dan ruang algoritma ini adalah $O(b^d)$, dengan b adalah *branching factor* dari graf dan d adalah jarak *node* tujuan dari *node* awal.

2.1.3. Iterative Deepening Search (IDS)

Iterative Deepening Search atau biasanya juga disebut *Iterative Deepening Depth First Search* adalah salah salah satu metode atau algoritma penjelajahan dalam graf yang merupakan bentuk pengembangan dari *Depth First Search* (DFS). Tidak seperti DFS yang melakukan pencarian secara mendalam terus-menerus hingga ditemukannya solusi atau hingga tidak ada lagi *node* yang bisa dikunjungi, dalam IDS kedalaman pencarian dibatasi dan ditingkatkan secara bertahap yang akan semakin mendalam seiring berjalannya waktu

dan pencarian, hingga mendapatkan *goal* yang diinginkan. Algoritma ini akan mengunjungi seluruh simpul pada kedalaman d sebelum melanjutkan pencarinya ke simpul $d + 1$. Kompleksitas waktu algoritma ini adalah $O(b^d)$ dan kompleksitas ruang algoritma ini adalah $O(bd)$, dengan b adalah *branching factor* dari graf dan d adalah jumlah kedalaman.

2.2. Penjelasan Web yang Dibangun

Hasil dari implementasi algoritma *Breadth First Search* dan *Iterative Deepening Search* dalam Wikipedia Race kami diimplementasikan dengan sebuah web. Website dapat menerima input judul wikipedia awal, judul wikipedia tujuan, serta pilihan algoritma yang ingin digunakan dan pilihan banyaknya solusi yang diinginkan (satu solusi atau banyak solusi).

Untuk mempermudah pengguna, website juga menyediakan *suggestion/saran* untuk setiap input yang pengguna berikan, dan memberikan *english wikipedia page* yang ada berdasarkan masukan user tersebut. Sebagai alternatif, pengguna juga dapat memasukkan *url english wikipedia page* secara langsung.

Setelah menemukan solusi dari Wikipedia Race, website akan menampilkan seluruh solusi yang didapatkan (atau hanya satu solusi jika memilih opsi satu solusi), jarak laman wikipedia tujuan ke laman wikipedia awal, jumlah laman yang dikunjungi selama proses pencarian, durasi waktu pencarian, serta menampilkan solusi yang didapatkan dalam bentuk graf.

BAB III

ANALISIS PEMECAHAN MASALAH

3.1. Langkah-Langkah Pemecahan Masalah

3.1.1. Pencarian Solusi dengan Metode *Breadth First Search* (BFS)

Pencarian solusi menggunakan BFS memanfaatkan struktur data *queue* untuk menyimpan setiap link yang akan diperiksa. Adapun langkah-langkah pencarian solusinya yaitu:

1. Pencarian dimulai dari laman awal. Masukkan laman awal ke dalam *queue*.
2. Kunjungi setiap isi *queue* saat ini, apakah mengandung laman tujuan atau tidak.
3. Jika laman tujuan terdapat pada isi *queue* saat ini, tambahkan laman tersebut ke dalam himpunan solusi.
4. Jika laman tujuan tidak terdapat pada isi *queue* saat ini, maka cari setiap link yang bertetangga dengan link-link yang ada pada *queue* saat ini namun belum pernah dikunjungi sebelumnya. Masukkan setiap link yang ditemukan ke dalam *queue* yang baru.
5. Catat juga laman asal dari masing-masing link tersebut saat proses pencarian link agar bisa melacak jalur dari laman awal ke link tersebut.
6. Ulangi langkah 2-5 hingga laman tujuan ditemukan.

Karena sifat BFS yang selalu mengunjungi *node* pada kedalaman d terlebih dahulu sebelum kedalaman $d+1$, maka proses pencarian *node* tetangga pada kedalaman d dapat dilakukan secara paralel untuk tiap-tiap *node*. Dengan memanfaatkan sifat ini, untuk bisa mempercepat proses pencarian, digunakan konsep *multithreading* dalam proses pencarian *node* tetangga dari tiap-tiap *node*.

Misal pada *queue* saat ini terdapat *node* A, B, C, dan D. Tanpa konsep *multithreading*, pencarian *node* tetangga akan dimulai dengan mencari *node-node* tetangga dari *node* A dan kemudian memasukkannya ke dalam *queue* yang baru. Setelah selesai, proses dilanjutkan dengan mencari *node-node* tetangga dari *node* B. Proses dilanjutkan hingga ke *node* D. Proses pencarian seperti ini tentu akan memakan cukup banyak waktu. Dengan memanfaatkan konsep *multithreading*, proses pencarian *node* tetangga dari *node* A, B, C, dan

D dapat dilakukan secara serentak tanpa harus menunggu *node* yang lain selesai. Dengan cara ini, proses pencarian dapat dilakukan lebih cepat.

3.1.2. Pencarian Solusi dengan Metode *Iterative Deepening Search* (IDS)

Proses pencarian menggunakan IDS dimulai dari tautan awal yang ingin di telusuri. Proses pencarian ini akan memanfaatkan algoritma pencarian *Depth Limited Search* (DLS). Adapun langkah-langkah pencarian dengan DLS yaitu:

1. Program akan mengecek apakah link yang sedang dimasukkan sekarang merupakan link tujuan. Apabila iya, jalur dari link awal ke link saat ini akan disimpan.
2. Jika tidak, akan dilakukan proses *scraping* pada tautan tersebut dan seluruh link yang terdapat dalam laman tersebut akan disimpan.
3. Untuk setiap link yang diperoleh, dilakukan kembali proses pada langkah 1-2 secara rekursif secara berurutan.
4. Basis untuk proses rekursif ini adalah saat kedalaman telah mencapai kedalaman maksimum yang diinginkan.

Adapun algoritma IDS hanya melakukan pemanggilan algoritma DLS secara berulang-ulang dimulai dari maksimum kedalaman satu dan dalam setiap perulangan dilakukan penambahan kedalaman jika solusi masih belum ditemukan.

Algoritma DLS yang digunakan memanfaatkan sifat struktur data *stack* dalam melakukan proses rekursif, dimana link yang baru dimasukkan beserta link-link di dalamnya akan terlebih dahulu di cek sebelum berpindah pada link berikutnya.

Karena sifat IDS yang selalu mengulang proses dari link awal setiap kali terjadi penambahan kedalaman, maka pada dasarnya sebagian *node* yang akan dikunjungi pada iterasi saat ini sebenarnya sudah pernah dikunjungi pada iterasi sebelumnya. Agar tidak dilakukan proses *scraping* web berulang-ulang pada link yang sama, maka hasil *scraping* web perlu disimpan agar bisa digunakan kembali setelahnya. Untuk itu, digunakanlah sistem *caching* pada algoritma IDS. Dengan *cache*, proses *scraping* web hanya perlu dilakukan sekali saat suatu link pertama kali dikunjungi. Jika pada iterasi selanjutnya link tersebut dikunjungi kembali, maka cukup dengan mengambil data pada *cache* yang ada sehingga dapat mempercepat proses pencarian solusi.

3.2. Pemetaan Masalah

Dalam program ini, setiap link merepresentasikan *node* dalam sebuah graf. Selain itu, untuk setiap *node*, disimpan juga informasi jalur dari artikel awal ke link (*node*) tersebut.

Pada algoritma BFS, setiap *node* akan disimpan ke dalam sebuah *queue*. Sedangkan pada algoritma IDS, setiap *node* akan disimpan ke dalam sebuah *stack*.

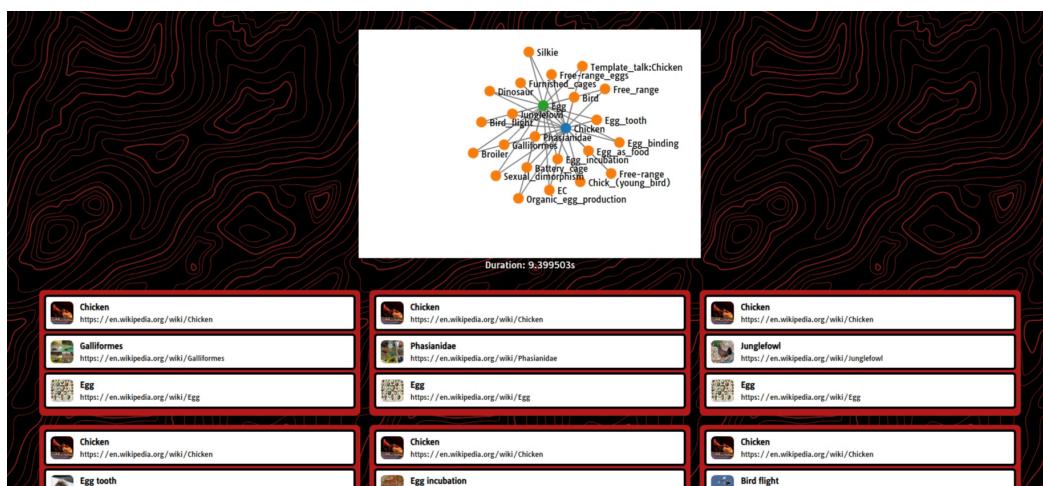
Untuk menyimpan informasi apakah sebuah *node* sudah pernah dikunjungi atau belum, digunakan struktur data *map* dengan link sebagai *key* dan nilai *boolean* (*true/false*) sebagai *value*. Jika sebuah *node* telah dikunjungi, maka *value node* tersebut di dalam *map* akan bernilai *true*. Jika sebuah *node* belum pernah dikunjungi, maka secara *default* *map* akan memberikan *value false*.

3.3. Fitur dan Arsitektur Website

3.3.1. Fitur-Fitur Website

1. Pembentukan Graf

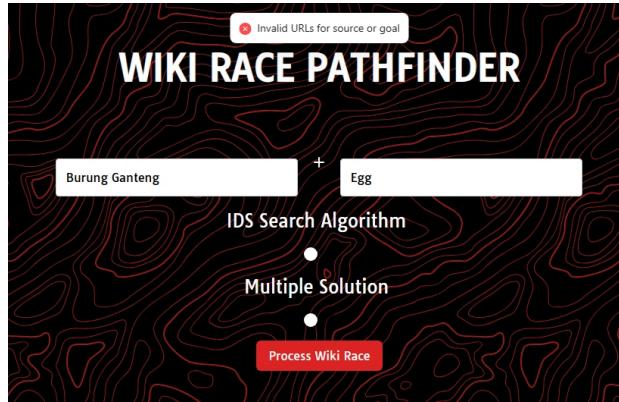
Hasil dari pencarian solusi yang diberikan kepada pengguna dapat disajikan dalam bentuk graf. Hal ini bertujuan untuk mempermudah visualisasi hubungan antara link awal dan link tujuan. Graf dibuat interaktif agar pengguna dapat memilih hubungannya secara bebas. Setiap simpul dari graf yang disajikan merupakan sebuah *hyperlink* yang akan mengalihkan pengguna ke laman dari simpul tersebut. Graf ini dibuat menggunakan library d3.js.



Gambar 2. Graf Solusi

2. Input Checking

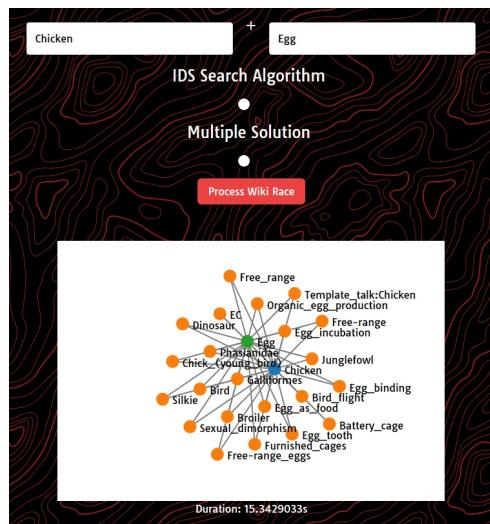
Dalam melakukan *input* pada *textbox*, sebelum program melakukan kalkulasi, website akan mengecek apakah judul dari artikel wikipedia merupakan judul yang valid dan ada di wikipedia itu sendiri. Apabila salah satu dari *input* ada yang tidak valid, akan dilakukan informasi *Invalid URL* dan pengguna dapat memasukkan kembali *input* hingga benar.



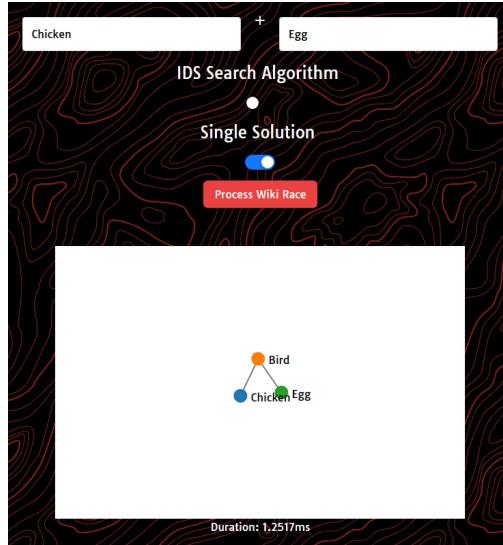
Gambar 3. Input Checking

3. Jumlah Solusi

Pengguna dapat memilih jumlah solusi yang diinginkan, baik itu banyak solusi maupun satu solusi. Keunggulan dari pemilihan solusi ini adalah waktu eksekusi yang lebih cepat. Hal ini dikarenakan pada algoritma pencarian, penelusuran dari link akan dihentikan apabila sudah terdapat satu solusi. Hal ini menyebabkan semakin sedikit link yang dikunjungi sehingga waktu pencarian dapat lebih cepat.



Gambar 4. Hasil Pencarian Solusi Banyak



Gambar 5. Hasil Pencarian Solusi Tunggal

Untuk memperoleh perbedaan jumlah solusi ini, telah dibuat suatu *endpoint API* terhadap *backend* yang unik sehingga *frontend* dapat, secara dinamis, mengubah permintaannya. *Endpoint* yang diciptakan adalah sebagai berikut:

```
...../{single|many}?source={source_title}&goal={goal_title}
```

4. Algoritma *Search Toggle*

Pengguna dapat memilih teknik penelusuran jalur dari dua metode, BFS atau IDS. Terdapat tombol bertipe *toggle* yang dapat ditekan oleh pengguna sehingga memberitahu program untuk melakukan *fetch* ke *endpoint* yang sesuai.



Gambar 6. Tombol Pemilihan Algoritma Pencarian

Endpoint dari API untuk melakukan BFS atau IDS merupakan sebagai berikut:

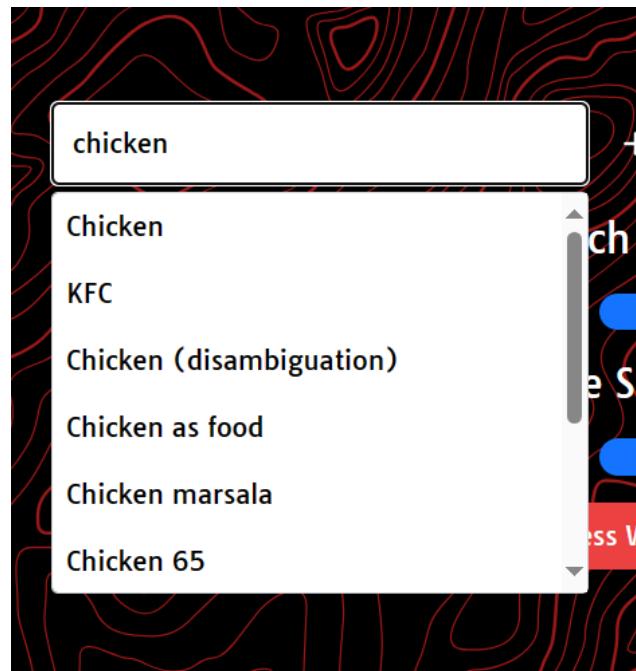
```
localhost:9090/{bfs|ids}/{single|many}?.....
```

3.3.2. Arsitektur Website

Aplikasi web ini mengadopsi arsitektur modern yang terdiri dari *frontend* dan *backend* yang terpisah. Bagian *frontend* dibangun dengan menggunakan React JS, sebuah *framework* JavaScript yang populer untuk pengembangan antarmuka pengguna yang responsif dan interaktif. Sementara itu, untuk *backend*, aplikasi ini mengandalkan *library* gin yang ditulis dalam bahasa pemrograman GO. Penggunaan kombinasi React JS dan gin memungkinkan aplikasi ini untuk menyajikan antarmuka yang dinamis di sisi klien sambil memberikan kinerja yang tangguh dan efisien di sisi server.

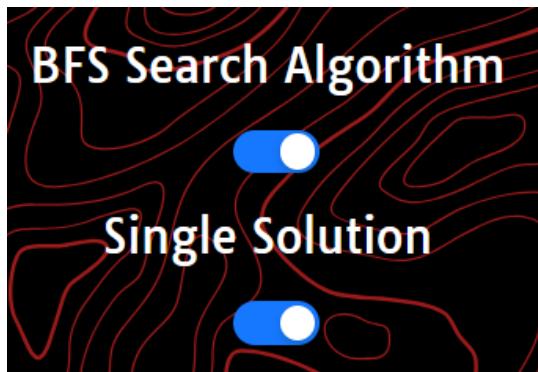
3.4. Ilustrasi Kasus

Pada awalnya pengguna akan diminta untuk memasukkan judul wikipedia awal dan judul wikipedia tujuan pada bagian *input* judul. Pada saat memasukkan judul wikipedia, laman web akan memberikan saran beberapa judul yang mendekati *input* pengguna saat ini. Adapun contohnya seperti pada gambar berikut.



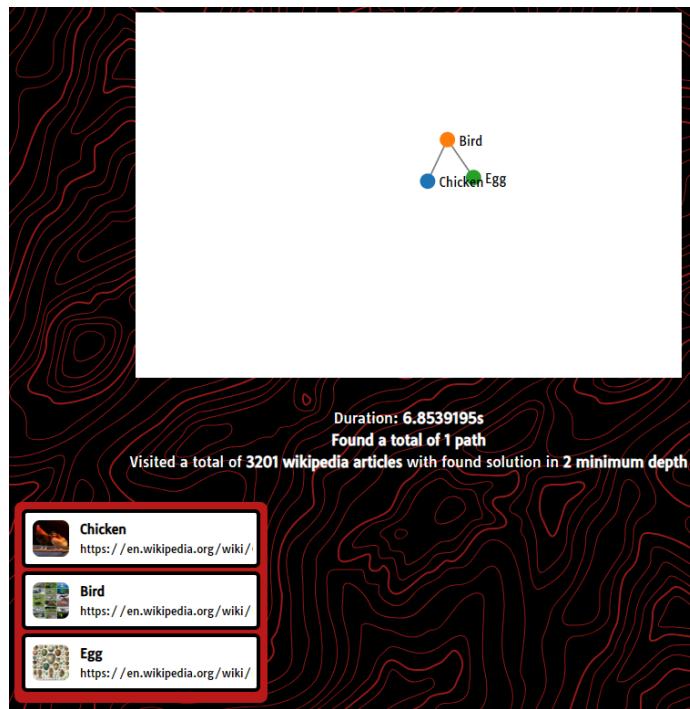
Gambar 7. Saran Judul Pencarian

Selanjutnya pengguna dapat memilih algoritma pencarian yang ingin digunakan serta jumlah solusi yang ingin ditampilkan dengan menekan *toggle button* yang disediakan. Berikut adalah contohnya.



Gambar 8. Koleksi Tombol

Selanjutnya, pengguna hanya perlu menekan tombol “*Process Wiki Race*” dan menunggu web menampilkan hasilnya. Setelah hasil keluar, web akan menampilkan graf solusi, durasi waktu pencarian, jarak dari judul wikipedia awal ke judul wikipedia tujuan, banyaknya laman artikel wikipedia yang dikunjungi selama pencarian, serta jalur dari setiap solusi yang ditemukan.



Gambar 9. Ilustrasi Tampilan Hasil Pencarian

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1. Spesifikasi Program

4.1.1. Struktur Data

1. Struct

Dalam implementasi algoritma, kami membuat dua buah *struct*, yaitu Struct Node dan Struct CacheFile.

```
src/scrapper/util.go

....  
  
// Define a struct to represent a node in the graph  
type Node struct {  
    Current string  
    Paths   []string  
}  
  
// Define a struct to represent a cache format in file  
type CacheFile struct {  
    Key      string `json:"key"  
    Neighbours []string `json:"Node"  
}  
  
....
```

Struct Node berperan sebagai representasi simpul dari data yang akan ditelusuri selama proses BFS maupun IDS. Struct ini menyimpan *string* dari link pada posisi saat ini dan larik dari *string* yang dibutuhkan untuk mencapai *node* saat ini dari link awal.

Struct CacheFile berperan sebagai struktur data untuk melakukan *cache* kedalam suatu file ataupun kebalikannya. Elemen “Key” akan menyimpan suatu link dan “Neighbours” akan menyimpan konten link yang terdapat di dalam Key.

2. Map

Penggunaan map dilakukan untuk:

- Melakukan pencatatan *node-node* yang sudah pernah dikunjungi selama proses pencarian solusi.

src/scrapper/util.go

....

```
// Define a map to keep track of node (link) that has been added to the
queue/stack
// if the node is added, the value is true, otherwise false
var Visited_Node = make(map[string]bool)
```

....

Link (bertipe *string*) bertindak sebagai *key* dari *map* dan *value*-nya berupa nilai boolean (*true/false*)

- Menyimpan cache dari suatu link.

src/scrapper/util.go

....

```
// Define a map to keep track of node (link) that has been added to the
queue/stack
// if the node is added, the value is true, otherwise false
var Cache = make(map[string][]node)
```

....

Link (bertipe *string*) bertindak sebagai *key* dari *map* dan *value*-nya merupakan *list of node* yang merupakan *node/link* tetangganya.

3. Array

Array digunakan untuk menyimpan daftar *node-node* selama proses pencarian (*queue* untuk BFS dan *stack* untuk IDS) dan daftar solusi yang diperoleh.

4.1.2. Fungsi dan Prosedur

1. Interface

Prosedur ini berfungsi untuk melakukan abstraksi terhadap *input* yang diperlukan untuk menjalankan fungsi algoritma utama (BFS/IDS).

a. IDS

src/scrapper/interface.go

```
func IDS_interface(link_awal string, link_tujuan string, solution_mode string) ([]Node, time.Duration) {
    printRequestedParameters(link_awal, link_tujuan)

    // read cache from file
    if len(Cache) == 0 {
        readCacheFromFile()
    }

    fmt.Println("Starting Iterative Deepening Search ")
    var solutions []Node

    // find the solution using Iterative Deepening Search (IDS) algorithm
    startTime := time.Now()
    if solution_mode == SINGLE_PARAM {
        fmt.Println("single solution..")
        iterative_deepening_search_single(link_awal, link_tujuan, &solutions)
    } else if solution_mode == MANY_PARAM {
        fmt.Println("multiple solution..")
        iterative_deepening_search_many(link_awal, link_tujuan, &solutions)
    }
    duration := time.Since(startTime)

    printSolution(solutions, duration)
    writeCacheToFile()

    return solutions, duration
}
```

b. BFS

```
src/scrapper/interface.go
```

```
func BFS_interface(link_awal string, link_tujuan string, solution_mode string) ([]Node,
time.Duration) {
    printRequestedParameters(link_awal, link_tujuan)
    fmt.Println("Starting Breadth First Search ")
    var initial = Node{
        Current: link_awal,
    }

    // find the solution using Breadth First Search (BFS) algorithm
    var solutions []Node
    startTime := time.Now()
    if solution_mode == SINGLE_PARAM {
        fmt.Println("single solution..")
        breadth_first_search_one_solution([]Node{initial}, link_tujuan, &solutions)
    } else if solution_mode == MANY_PARAM {
        fmt.Println("multiple solution..")
        breadth_first_search_many_solution([]Node{initial}, link_tujuan,
&solutions)
    }
    duration := time.Since(startTime)

    printSolution(solutions, duration)

    // clean map Visited_Node
    Visited_Node = make(map[string]bool)

    return solutions, duration
}
```

2. Depth Limited Search (DLS)

Merupakan logika utama dari implementasi *Depth Limited Search*. Implementasi ini dibagi menjadi dua.

a. Satu Solusi

```
src/scrapper/ids.go
```

```
func depth_limited_search_one_solution(currentNode Node, destinationLink string, depth int,
hasil *[]Node, path []string) {
    if len(*hasil) > 0 {
        return
    } else if depth == 0 { // if the current node is the deepest node
        if currentNode.Current == destinationLink {
            currentNode.Paths = path
            *hasil = append(*hasil, currentNode)
        }
    } else {
        path = append(path, currentNode.Current)
        var tetangga []Node

        // get the adjacent links from the current node
        if isInCache(currentNode.Current) {
            tetangga = Cache[currentNode.Current]
        } else {
            tetangga = getAdjacentLinks(currentNode, "ids")
            Cache[currentNode.Current] = tetangga
        }

        // go through all the adjacent links
        for _, node := range tetangga {
            Total_Visited_Link++
            depth_limited_search_one_solution(node, destinationLink, depth-1,
hasil, path)
            if len(*hasil) > 0 {
                return
            }
        }
    }
}
```

b. Banyak Solusi

```
src/scrapper/ids.go
```

```
func depth_limited_search_many_solution(currentNode Node, destinationLink string, depth
int, hasil *[]Node, path []string) {
    if depth == 0 { // if the current node is the deepest node
        if currentNode.Current == destinationLink {
            currentNode.Paths = path
            *hasil = append(*hasil, currentNode)
        }
    } else {
        path = append(path, currentNode.Current)
        var tetangga []Node

        // get the adjacent links from the current node
        if isInCache(currentNode.Current) {
            tetangga = Cache[currentNode.Current]
        } else {
            tetangga = getAdjacentLinks(currentNode, "ids")
            Cache[currentNode.Current] = tetangga
        }

        // go through all the adjacent links
        for _, node := range tetangga {
            Total_Visited_Link++
            depth_limited_search_many_solution(node, destinationLink, depth-1,
hasil, path)
        }
    }
}
```

3. Iterative Deepening Search (IDS)

Merupakan logika utama dari implementasi *Iterative Deepening Search* memanfaatkan prosedur *Depth Limited Search*. Implementasi ini dibagi menjadi dua.

- Satu Solusi

```
src/scrapper/ids.go
```

```
func iterative_deepening_search_single(startLink string, destinationLink string, hasil
*[]Node) {
    var initial = Node{
        Current: startLink,
    }

    var solutions []Node

    // check if the start node is the destination node
    if initial.Current == destinationLink {
        solutions = append(solutions, initial)
    }

    // start the depth from 1
    depth := 1
    for len(solutions) == 0 {
        Total_Visited_Link = 0
        fmt.Println("Starting new depth..")
        depth_limited_search_one_solution(initial, destinationLink, depth,
&solutions, []string{})
        depth++
    }

    *hasil = solutions
}
```

b. Banyak Solusi

```
src/scrapper/ids.go
```

```
func iterative_deepening_search_many(startLink string, destinationLink string, hasil
*[]Node) {
    var initial = Node{
        Current: startLink,
    }

    var solutions []Node

    // check if the start node is the destination node
    if initial.Current == destinationLink {
        solutions = append(solutions, initial)
    }

    // start the depth from 1
    depth := 1
    for len(solutions) == 0 {
        Total_Visited_Link = 0
        fmt.Println("Starting new depth..")
        depth_limited_search_many_solution(initial, destinationLink, depth,
&solutions, []string{})
        depth++
    }

    *hasil = solutions
}
```

4. Breadth First Search (BFS)

Merupakan logika utama dari implementasi *Breadth First Search*. Implementasi ini dibagi menjadi dua.

a. Satu Solusi

```
src/scrapper/ids.go
```

```
func breadth_first_search_one_solution(currentQueue []Node, destinationLink string, hasil *[]Node) {
    if len(currentQueue) == 0 {
        return
    } else {
        // go through all the nodes in the current queue
        // check if the destination node is in the current queue
        for _, current_node := range currentQueue {
            Visited_Node[current_node.Current] = true
            Total_Visited_Link++
            if current_node.Current == destinationLink {
                *hasil = append(*hasil, current_node)
                return
            }
        }

        // if the current queue does not have the destination node
        // create a goroutine for each link to concurrently get the adjacent links
        var wg sync.WaitGroup

        // make a new queue to store the adjacent links from the current queue
        var newQueue []Node

        // process queue in blocks of THREADS to avoid spam http request
        startIndeks := 0
        endIndeks := THREADS
        for endIndeks < len(currentQueue) {
            // limit the number of goroutines to THREADS to avoid spam http
            request
            wg.Add(THREADS)

            for i := startIndeks; i < endIndeks; i++ {
                go func(current_node Node) {
                    defer wg.Done()
                    // get the adjacent links from the current node
                    adjacentLinks := getAdjacentLinks(current_node,
                        "bfs")
                }
            }
        }
    }
}
```

```

        newQueue = append(newQueue, adjacentLinks...)
    }(currentQueue[i])
}

wg.Wait()
startIndeks += THREADS
endIndeks += THREADS
}

// process the remaining links
wg.Add(len(currentQueue) - startIndeks)
for i := startIndeks; i < len(currentQueue); i++ {
    go func(current_node Node) {
        defer wg.Done()
        // get the adjacent links from the current node
        adjacentLinks := getAdjacentLinks(current_node, "bfs")
        newQueue = append(newQueue, adjacentLinks...)
    }(currentQueue[i])
}
// wait for all goroutines to finish
wg.Wait()

// recursively call the breadth_first_search function with the new queue
breadth_first_search_one_solution(newQueue, destinationLink, hasil)
}
}

```

b. Banyak Solusi

```
src/scrapper/ids.go
```

```
func breadth_first_search_many_solution(currentQueue []Node, destinationLink string, hasil *[]Node) {
    if len(currentQueue) == 0 {
        return
    } else {
        // go through all the nodes in the current queue
        // check if the destination node is in the current queue
        isDestinationLinkExist := false
        for _, current_node := range currentQueue {
            Visited_Node[current_node.Current] = true
            if current_node.Current == destinationLink {
                *hasil = append(*hasil, current_node)
                isDestinationLinkExist = true
            }
        }
        Total_Visited_Link += len(currentQueue)

        if isDestinationLinkExist {
            // solution is found, return
            return
        } else {
            // if the current queue does not have the destination node
            // create a goroutine for each link to concurrently get the
            adjacent links
            var wg sync.WaitGroup

            // make a new queue to store the adjacent links from the current
            queue
            var newQueue []Node

            // process queue in blocks of THREADS to avoid spam http request
            startIndeks := 0
            endIndeks := THREADS
            for endIndeks < len(currentQueue) {
                // limit the number of goroutines to THREADS to avoid spam
                http request
                wg.Add(THREADS)

                for i := startIndeks; i < endIndeks; i++ {
                    go func(current_node Node) {
                        defer wg.Done()

```

```

        // get the adjacent links from the current
node
        adjacentLinks :=
getAdjacentLinks(current_node, "bfs")
        newQueue = append(newQueue,
adjacentLinks...)
        }(currentQueue[i])
    }

wg.Wait()
startIndeks += THREADS
endIndeks += THREADS
}

// process the remaining links
wg.Add(len(currentQueue) - startIndeks)
for i := startIndeks; i < len(currentQueue); i++ {
    go func(current_node Node) {
        defer wg.Done()
        // get the adjacent links from the current node
        adjacentLinks := getAdjacentLinks(current_node,
"bfs")
        newQueue = append(newQueue, adjacentLinks...)
        }(currentQueue[i])
    }
    // wait for all goroutines to finish
    wg.Wait()

    // recursively call the breadth_first_search function with the new
queue
    breadth_first_search_many_solution(newQueue, destinationLink,
hasil)
}
}
}

```

5. Is In Cache

Merupakan sebuah fungsi yang mengembalikan tipe data *boolean* untuk mengecek apakah suatu link sudah ada di *cache* atau belum.

```
src/scrapper/util.go
```

```
func isInCache(link string) bool {
    return len(Cache[link]) != 0
}
```

6. Get Adjacent Links

Merupakan fungsi yang mengembalikan tipe data larik Node yang diperoleh melalui *scraping* dari suatu website.

```
src/scrapper/util.go
```

```
func getAdjacentLinks(activeNode Node, mode string) []Node {
    // Get the HTML document from the current link
    link := DOMAIN_PREFIX + activeNode.Current
    res := getResponse(link)
    doc := getDocument(res)

    // If the document is nil, return an empty list
    if doc == nil {
        return []Node{}
    } else {
        unique := make(map[string]bool) // to make sure that the link is
        unique

        // Find all the links in the HTML document
        var linkNodes []Node
        // Filter the links that start with "/wiki/" and do not contain
        "."

        doc.Find("div.mw-content-ltr.mw-parser-output").Find("a").FilterFunction(stringFilter).Each(
            func(i int, s *goquery.Selection) {
                linkTemp, _ := s.Attr("href")
```

```

        // Create a new node for each link
        var tempNode Node
        tempNode.Current = removeHash(linkTemp)

        if mode == "bfs" {
            tempNode.Paths = append(activeNode.Paths,
removeHash(activeNode.Current))
        }

        // Append the new node to the list of links
        if !Visited_Node[tempNode.Current] &&
!unique[tempNode.Current] {
            linkNodes = append(linkNodes, tempNode)
            unique[tempNode.Current] = true
        }
    },
}

return linkNodes
}
}

```

4.2. Cara Penggunaan Program

4.2.1. Front-End

1. Buka terminal pada perangkat.
2. Lakukan *clone repository* dengan menjalankan perintah berikut:

```
git clone https://github.com/Agil0975/Tubes2_FE_Penyelam_Handal.git
```

3. Masuk ke folder Tubes2_FE_Penyelam_Handal.

```
cd Tubes2_FE_Penyelam_Handal
```

4. Lakukan instalasi dependensi.

```
npm install
```

5. Jalankan program.

```
npm run start
```

4.2.2. Back-End

1. Buka terminal pada perangkat.
2. Lakukan *clone repository* dengan menjalankan perintah berikut:

```
git clone https://github.com/raflyhangga/Tubes2_BE_Penyelam_Handal.git
```

3. Masuk ke folder Tubes2_BE_Penyelam_Handal dan folder src.

```
cd Tubes2_BE_Penyelam_Handal/src
```

4. Jalankan perintah berikut

```
go mod tidy
go get -u github.com/gin-gonic/gin
go get github.com/gin-contrib/cors
go get github.com/PuerkitoBio/goquery
```

5. Jalankan program.

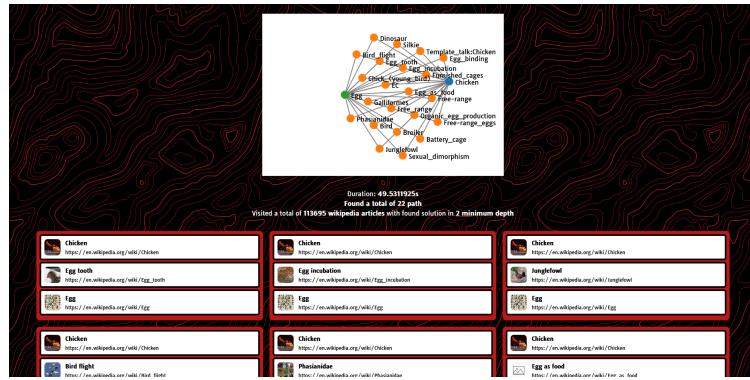
```
go run .
```

4.2.3. Menjalankan Permainan Wiki Race



Gambar 10. Laman Utama Program

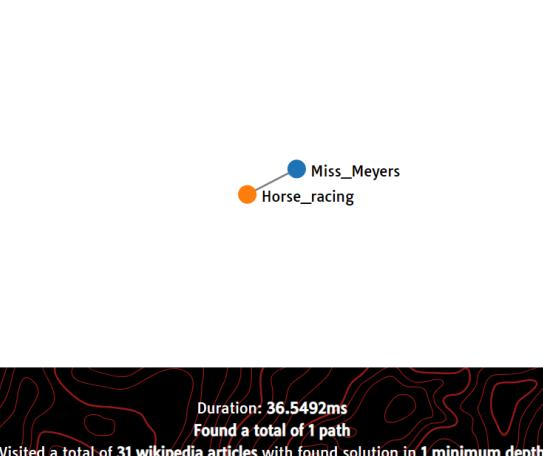
1. Pertama, pengguna memasukkan judul artikel wikipedia awal dan judul artikel wikipedia tujuan. Pada contoh di gambar, *Chicken* menjadi judul artikel awal dan *Football* menjadi judul artikel tujuan.
2. Setelah itu, pengguna dapat memilih algoritma pencarian graf yang ingin digunakan dengan menekan *toggle button* yang tersedia. Aktifkan *toggle button* untuk memilih opsi algoritma BFS dan matikan *toggle button* untuk memilih opsi IDS.
3. Selanjutnya pengguna memilih jumlah solusi yang ingin dihasilkan dengan menekan *toggle button* yang tersedia. Aktifkan *toggle button* untuk memilih opsi satu solusi dan matikan *toggle button* untuk memilih opsi banyak solusi.
4. Terakhir pengguna hanya perlu menekan tombol “*Process Wiki Race*” dan program akan mulai melakukan pencarian solusi. Pengguna hanya perlu menunggu hingga program menampilkan hasilnya di laman web. Berikut merupakan contoh hasil pencarian untuk inputan pada gambar berikut.



Gambar 11. Output Program

4.3. Hasil Pengujian

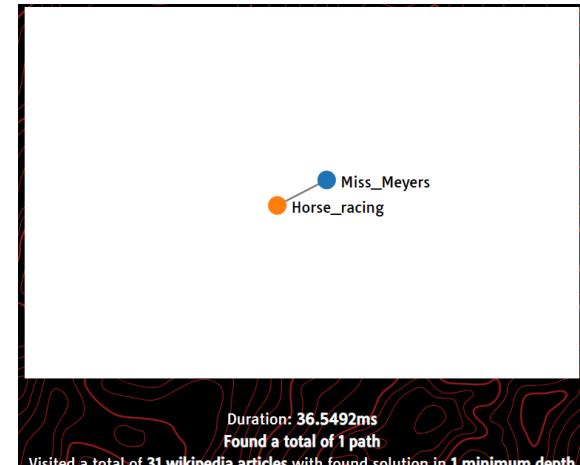
4.3.1. Breadth First Search (BFS)

| | |
|-------------|---|
| Kedalaman 1 |  <p>Duration: 34.5795ms Found a total of 1 path Visited a total of 3 wikipedia articles with found solution in 1 minimum depth</p> |
| |  <p>Duration: 36.5492ms Found a total of 1 path Visited a total of 31 wikipedia articles with found solution in 1 minimum depth</p> |

Gambar 12. Hasil Test Case 1 BFS Satu Solusi

Artikel Awal: Miss Meyers

Artikel Tujuan: Horse Racing

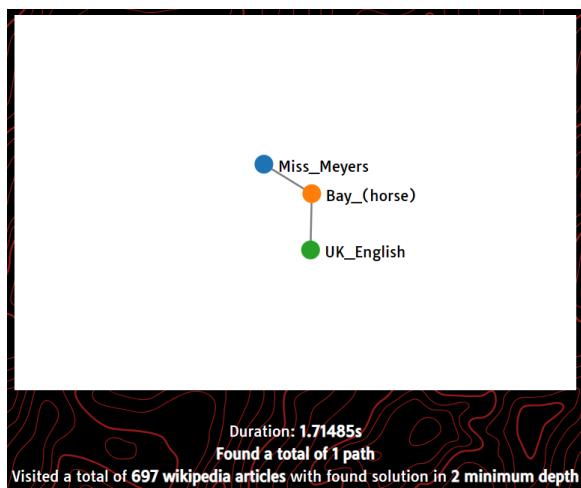


Gambar 13. Hasil Test Case 1 BFS Banyak Solusi

Artikel Awal: Miss Meyers

Artikel Tujuan: Horse Racing

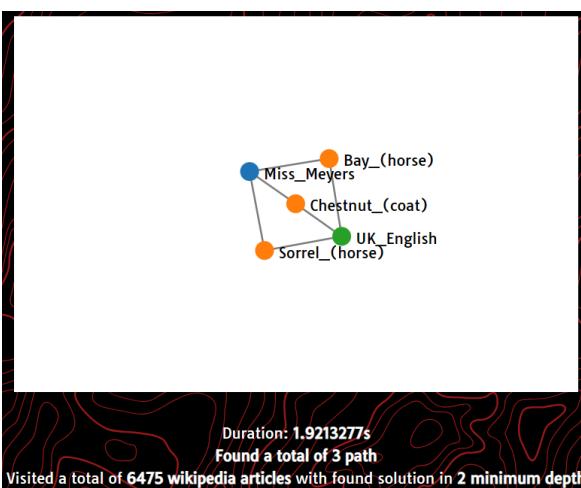
Kedalaman 2



Gambar 14. Hasil *Test Case 2 BFS* Satu Solusi

Artikel Awal: Miss Meyers

Artikel Tujuan: UK English

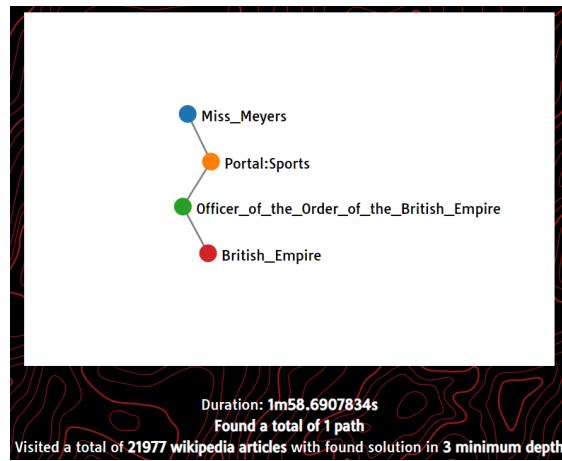


Gambar 15. Hasil *Test Case 2 BFS* Banyak Solusi

Artikel Awal: Miss Meyers

Artikel Tujuan: UK English

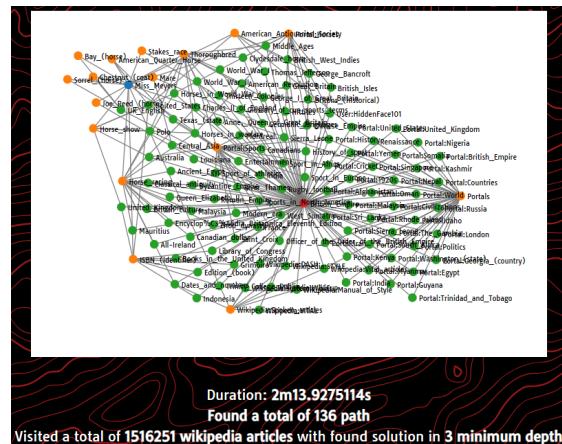
Kedalaman 3



Gambar 16. Hasil *Test Case 3 BFS* Satu Solusi

Artikel Awal: Miss Meyers

Artikel Tujuan: British Empire



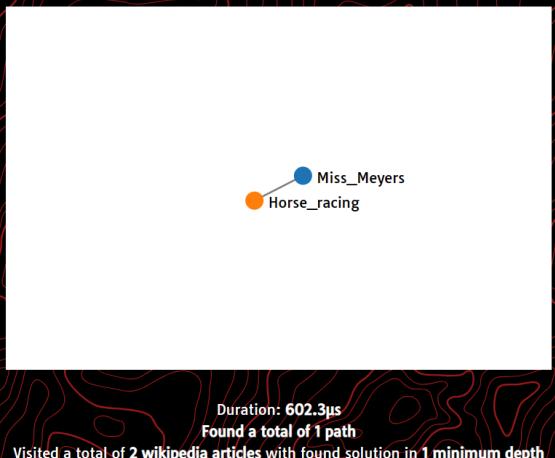
Gambar 17. Hasil *Test Case 3 BFS* Banyak Solusi

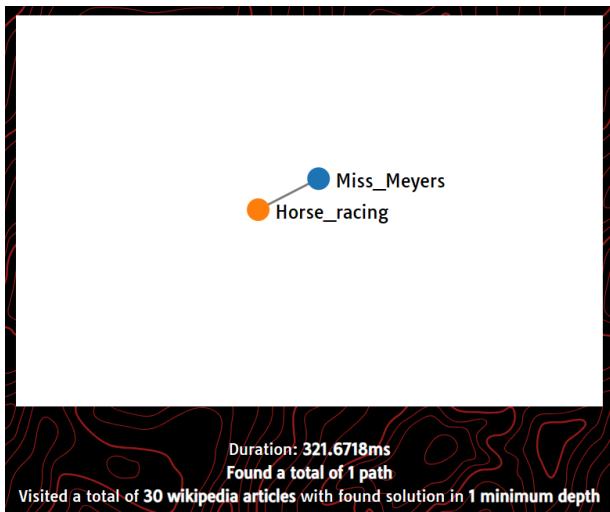
Artikel Awal: Miss Meyers

Artikel Tujuan: British Empire

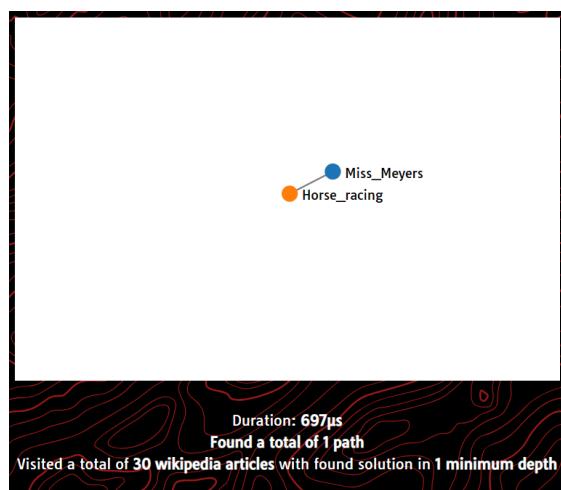
Tabel 1. Tabel Hasil Uji Coba Algoritma BFS

4.3.2. Iterative Deepening Search (IDS)

| | |
|-------------|---|
| Kedalaman 1 |  <p>A screenshot of a search interface. At the top, there are two nodes: a blue circle labeled "Miss_Meyers" and an orange circle labeled "Horse_racing". A line connects them, representing a path. Below the interface is a black bar with white text: "Duration: 602.3us", "Found a total of 1 path", and "Visited a total of 2 wikipedia articles with found solution in 1 minimum depth".</p> |
| | <p>Gambar 18. Hasil Test Case 1 IDS Satu Solusi tanpa Cache Artikel Awal: Miss Meyers Artikel Tujuan: Horse Racing</p> <p>Gambar 19. Hasil Test Case 1 IDS Satu Solusi dengan Cache Artikel Awal: Miss Meyers Artikel Tujuan: Horse Racing</p> |

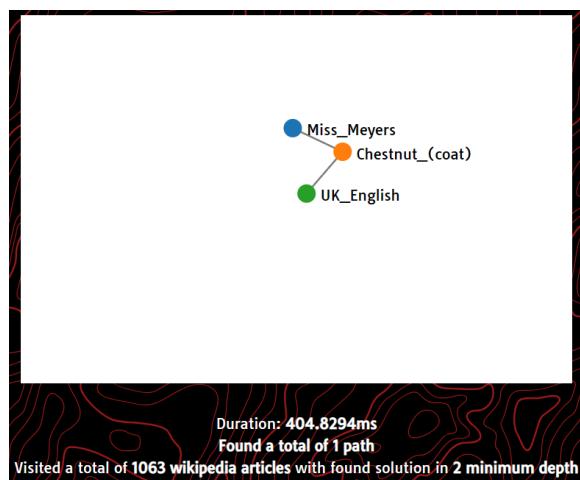


Gambar 20. Hasil *Test Case 1* IDS Banyak Solusi tanpa *Cache*
Artikel Awal: Miss Meyers
Artikel Tujuan: Horse Racing



Gambar 21. Hasil *Test Case 1* IDS Banyak Solusi dengan *Cache*
Artikel Awal: Miss Meyers
Artikel Tujuan: Horse Racing

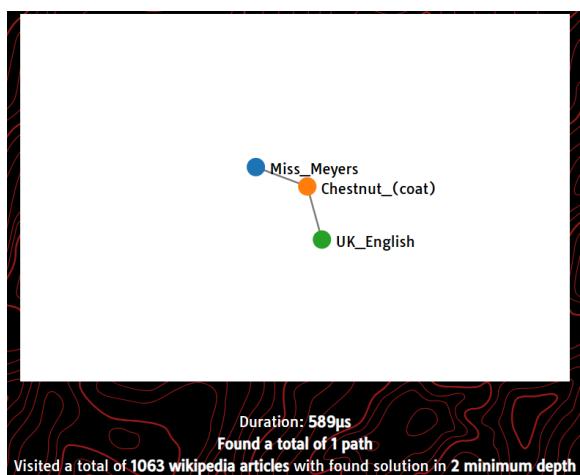
Kedalaman 2



Gambar 22. Hasil *Test Case 2* IDS Satu Solusi tanpa *Cache*

Artikel Awal: Miss Meyers

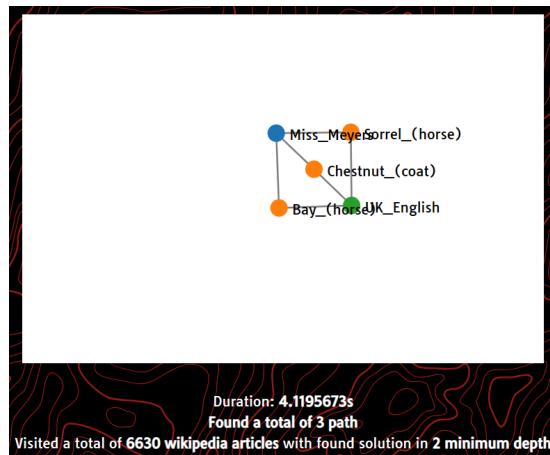
Artikel Tujuan: UK English



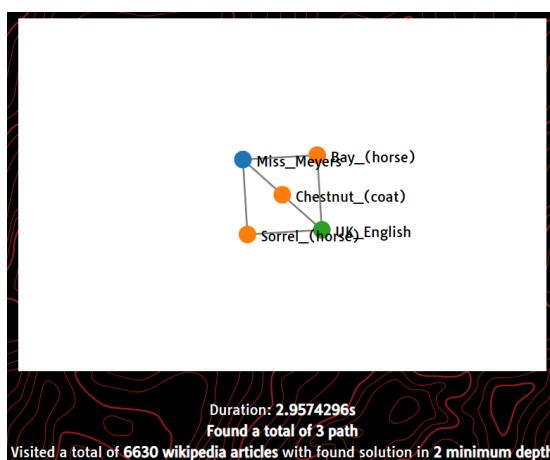
Gambar 23. Hasil *Test Case 2* IDS Satu Solusi dengan *Cache*

Artikel Awal: Miss Meyers

Artikel Tujuan: UK English

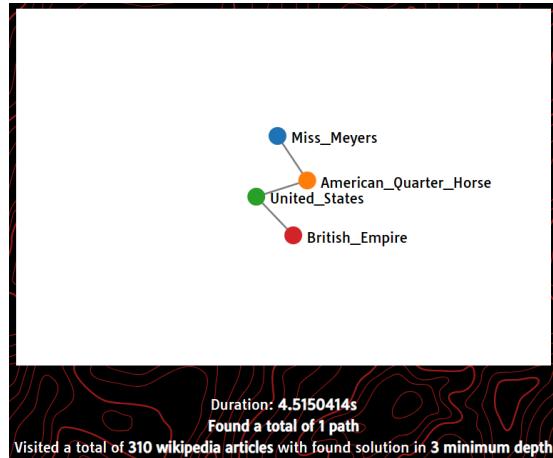


Gambar 24. Hasil Test Case 2 IDS Banyak Solusi tanpa Cache
 Artikel Awal: Miss Meyers
 Artikel Tujuan: UK English



Gambar 25. Hasil Test Case 2 IDS Banyak Solusi dengan Cache
 Artikel Awal: Miss Meyers
 Artikel Tujuan: UK English

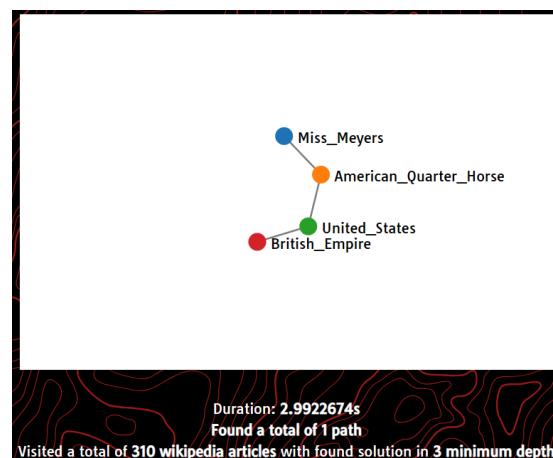
Kedalaman 3



Gambar 26. Hasil Test Case 3 IDS Satu Solusi tanpa Cache

Artikel Awal: Miss Meyers

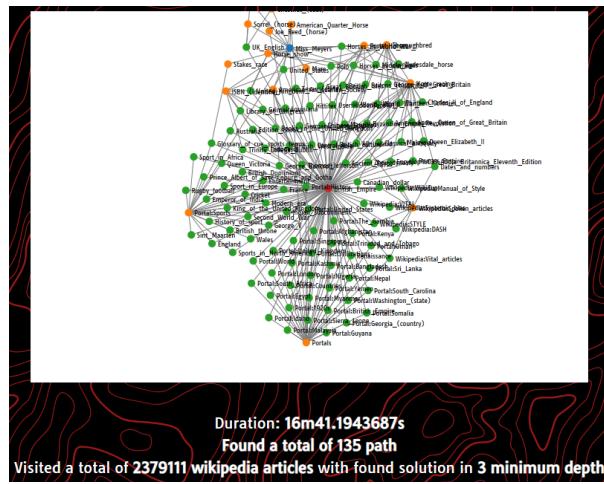
Artikel Tujuan: British Empire



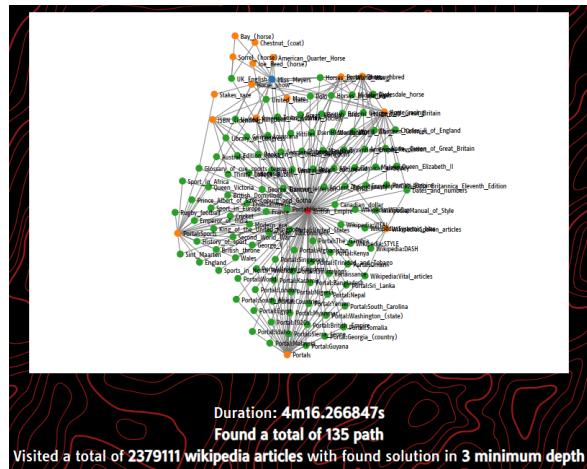
Gambar 27. Hasil Test Case 3 IDS Satu Solusi dengan Cache

Artikel Awal: Miss Meyers

Artikel Tujuan: British Empire



Gambar 28. Hasil Test Case 3 IDS Banyak Solusi tanpa Cache
 Artikel Awal: Miss Meyers
 Artikel Tujuan: British Empire



Gambar 29. Hasil Test Case 3 IDS Banyak Solusi dengan Cache
 Artikel Awal: Miss Meyers
 Artikel Tujuan: British Empire

Tabel 2. Tabel Hasil Uji Coba Algoritma IDS

4.4. Analisis Hasil Pengujian

4.4.1. Analisis Hasil Pengujian Algoritma BFS

Berdasarkan hasil pengujian yang dilakukan pada sub bab 4.3, terlihat bahwa durasi waktu pencarian serta jumlah link yang dikunjungi pada setiap kedalaman yang dicoba meningkat secara drastis dari kedalaman sebelumnya.

Pada percobaan dengan kedalaman satu, dilakukan pencarian dari “Miss Meyers” ke “Horse Racing”. Pada pencarian satu solusi, diperlukan waktu pencarian selama 0,034 detik dengan total kunjungan sebanyak 3 artikel wikipedia. Pada pencarian banyak solusi, diperoleh hanya satu solusi yang memerlukan waktu pencarian selama 0,036 detik dengan total kunjungan sebanyak 31 artikel wikipedia.

Pada percobaan dengan kedalaman dua, dilakukan pencarian dari “Miss Meyer” ke “UK English”. Pada pencarian satu solusi, diperlukan waktu pencarian selama 1,715 detik dengan total kunjungan sebanyak 697 artikel wikipedia. Pada pencarian banyak solusi, diperoleh 3 solusi yang memerlukan waktu pencarian selama 1,921 detik dengan total kunjungan sebanyak 6.475 artikel wikipedia.

Pada percobaan dengan kedalaman dua, dilakukan pencarian dari “Miss Meyer” ke “British Empire”. Pada pencarian satu solusi, diperlukan waktu pencarian selama 118,691 detik dengan total kunjungan sebanyak 21.977 artikel wikipedia. Pada pencarian banyak solusi, diperoleh 132 solusi yang memerlukan waktu pencarian selama 133,928 detik dengan total kunjungan sebanyak 1.516.251 artikel wikipedia.

Adapun detailnya dapat dilihat pada tabel di bawah ini.

| Kedalaman Pencarian | Satu Solusi | Banyak Solusi |
|---------------------|-------------|---------------|
| 1 | 0,034 s | 0,036 s |
| 2 | 1,715 s | 1,921 s |
| 3 | 118,691 s | 133,928 s |

Tabel 3. Waktu Eksekusi BFS dalam detik

| Kedalaman Pencarian | Satu Solusi | Banyak solusi |
|---------------------|----------------|-------------------|
| 1 | 3 artikel | 31 artikel |
| 2 | 697 artikel | 6.475 artikel |
| 3 | 21.977 artikel | 1.516.251 artikel |

Tabel 4. Banyak Artikel yang Dikunjungi Selama Pencarian BFS

Dari penjelasan sebelumnya terlihat bahwa waktu penemuan solusi meningkat secara eksponensial. Waktu pencarian solusi pada kedalaman dua 50,44 kali lebih lama dari pada kedalaman satu untuk pencarian satu solusi dan 53,36 kali lebih lama untuk pencarian banyak solusi. Adapun waktu pencarian solusi pada kedalaman tiga 69,21 kali lebih lama dari pada kedalaman dua untuk pencarian satu solusi dan 69,71 kali lebih lama untuk pencarian banyak solusi. Hal ini sesuai dengan kompleksitas waktu pencarian BFS yang bersifat eksponensial, yaitu $O(b^d)$, dengan b menyatakan rata-rata banyaknya *node* tetangga dari sebuah node dan d menyatakan kedalaman maksimum pencarian.

Adapun untuk kompleksitas ruang, banyaknya artikel wikipedia yang dilalui pada pencarian satu solusi pada kedalaman dua 232 kali lebih banyak dari pada kedalaman satu untuk pencarian satu solusi dan 209 kali lebih banyak untuk pencarian banyak solusi. Adapun banyaknya artikel wikipedia yang dilalui pada pencarian solusi pada kedalaman tiga 31 kali lebih lama dari pada kedalaman dua untuk pencarian satu solusi dan 234 lebih banyak untuk pencarian banyak solusi. Hal ini sesuai dengan kompleksitas ruang pencarian BFS yang bersifat eksponensial, yaitu $O(b^d)$, yang dihitung berdasarkan kemungkinan terburuk (mengunjungi semua *node* yang ada atau sama artinya dengan mencari semua kemungkinan solusi).

Akan tetapi, dapat dilihat bahwa durasi waktu pencarian untuk satu solusi maupun banyak solusi tidak terdapat perbedaan yang signifikan. Hal ini terjadi karena algoritma program yang dibuat untuk pencarian satu solusi hampir sama dengan algoritma pencarian banyak solusi. Baik pada pencarian satu solusi maupun banyak solusi, algoritma program akan melakukan proses *scraping* web pada setiap artikel

wikipedia yang ada pada kedalaman d sebelum melakukan pengecekan setiap artikel pada kedalaman $d+1$. Akibatnya, waktu untuk proses ekstraksi link tidak akan terdapat perbedaan kecuali dari faktor kecepatan internet saat proses pencarian berjalan. Satu-satunya yang berpengaruh pada perbedaan durasi waktu pencarian adalah pada saat pemeriksaan setiap node di kedalaman $d+1$ apakah merupakan artikel tujuan atau tidak. Pada pencarian satu solusi, proses akan berhenti saat pertama kali bertemu dengan artikel tujuan. Sedangkan pada pencarian banyak solusi, proses akan diteruskan hingga semua artikel diperiksa. Namun karena kecepatan pemrosesan dari bahasa Golang, perbedaan waktu ini menjadi tidak terlalu besar. Oleh karena itulah durasi waktu pencarian satu solusi tidak berbeda jauh dari pencarian banyak solusi.

4.4.2. Analisis Hasil Pengujian Algoritma IDS

Berdasarkan hasil pengujian yang dilakukan pada subbab 4.3, terdapat perbedaan waktu proses pencarian solusi yang signifikan antara IDS satu solusi dengan banyak solusi serta antara IDS dengan *cache* dengan tanpa *cache*. Perbedaan ini terjadi dikarenakan adanya perbedaan cara untuk mengambil link tetangga serta banyaknya kunjungan yang dilakukan oleh algoritma.

Pada percobaan dengan kedalaman satu, dilakukan pencarian solusi tunggal dari “Miss Meyers” ke “Horse Racing”. Pada pencarian tanpa adanya *cache*, diperlukan waktu pencarian selama 0,231 detik dengan total kunjungan sebanyak 2 artikel wikipedia. Pada pencarian dengan adanya *cache*, diperoleh waktu 602 mikrodetik dengan total kunjungan sebanyak 2 artikel wikipedia.

Pada pencarian banyak solusi, pada pencarian tanpa adanya *cache*, diperlukan waktu pencarian selama 0,321 s dengan total kunjungan sebanyak 30 artikel wikipedia. Pada pencarian dengan adanya *cache*, diperlukan waktu 697 mikrodetik dengan total kunjungan sebanyak 30 artikel wikipedia.

Pada percobaan dengan kedalaman dua, dilakukan pencarian solusi tunggal dari “Miss Meyer” ke “UK English”. Pada pencarian tanpa adanya *cache*, diperlukan waktu pencarian selama 0,404 detik dengan total kunjungan sebanyak 1.063 artikel wikipedia. Pada

pencarian dengan adanya *cache*, diperlukan waktu 589 mikrodetik dengan total kunjungan sebanyak 1.063 artikel wikipedia.

Pada pencarian solusi banyak, pada pencarian tanpa adanya *cache*, diperlukan waktu pencarian selama 4,12 detik dengan total kunjungan sebanyak 6.630 artikel wikipedia. Pada pencarian dengan adanya *cache*, diperlukan waktu 2,95 detik dengan total kunjungan sebanyak 6.630 artikel wikipedia.

Pada percobaan dengan kedalaman tiga, dilakukan pencarian solusi tunggal dari “Miss Meyers” ke “British Empire”. Pada pencarian tanpa adanya *cache*, diperlukan waktu pencarian selama 4,52 detik dengan total kunjungan sebanyak 310 artikel wikipedia. Pada pencarian dengan adanya *cache*, diperlukan waktu 2,99 detik dengan total kunjungan sebanyak 310 artikel wikipedia.

Pada pencarian solusi banyak, pada pencarian tanpa adanya *cache*, diperlukan waktu pencarian selama 16 menit 41 detik dengan total kunjungan sebanyak 2.379.111 artikel wikipedia. Pada pencarian dengan adanya *cache*, diperlukan waktu 4 menit 16 detik dengan total kunjungan sebanyak 2.379.111 artikel wikipedia.

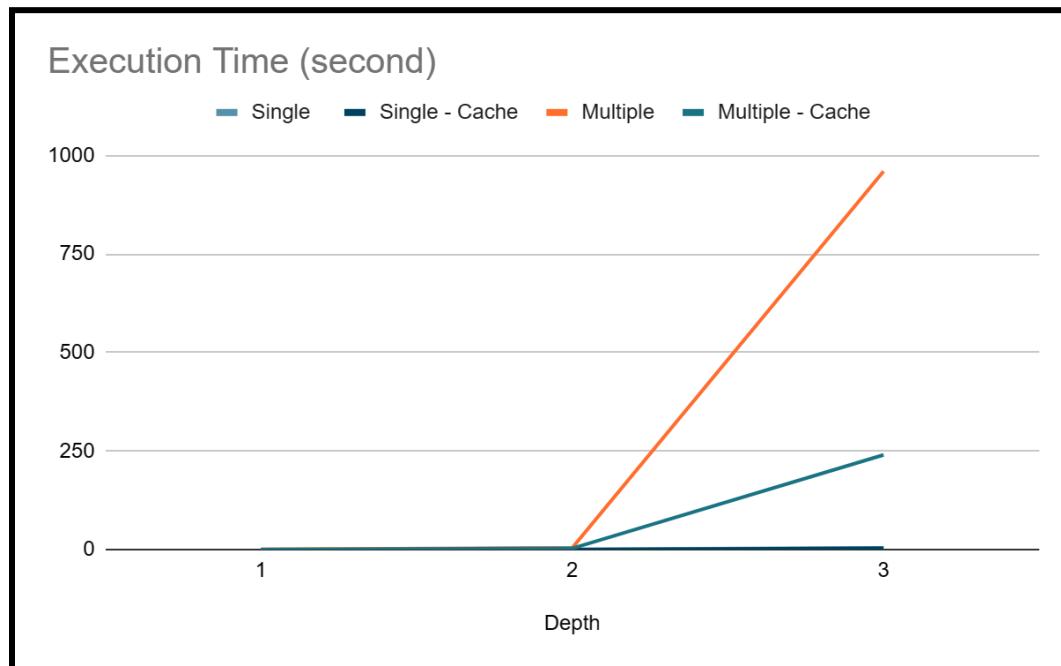
Adapun detailnya dapat dilihat pada tabel dibawah ini.

| Kedalaman | Satu Solusi | | Banyak Solusi | |
|-----------|-------------|----------------|---------------|----------------|
| | Tanpa Cache | Dengan Cache | Tanpa Cache | Dengan Cache |
| 1 | 0.231 detik | 602 mikrodetik | 0.321 detik | 697 mikrodetik |
| 2 | 0.404 detik | 589 mikrodetik | 4.12 detik | 2.95 detik |
| 3 | 4.52 detik | 2.99 detik | 1001 detik | 256 detik |

Tabel 5. Waktu Eksekusi IDS dalam detik

| Kedalaman Pencarian | Satu Solusi | Banyak solusi |
|---------------------|---------------|-------------------|
| 1 | 2 artikel | 30 artikel |
| 2 | 1.063 artikel | 6.630 artikel |
| 3 | 310 artikel | 2.379.111 artikel |

Tabel 6. Banyak Artikel yang Dikunjungi Selama Pencarian BFS



Gambar 30. Grafik *Execution Time* IDS

Hasil uji berdasarkan waktu pencarian solusi telah digambarkan dalam grafik di atas. Kesimpulan yang bisa diambil adalah semakin dalam proses pencarian, maka semakin lama waktu eksekusinya. Hal ini bersesuaikan dengan kompleksitas waktu dari *Iterative Deepening Search* itu sendiri yaitu $O(b^d)$ dengan b merupakan maksimum tetangga dari suatu link serta d menyatakan kedalaman maksimum pencarian. Dari segi perubahan waktu, dipastikan tiap kedalaman memiliki perbedaan yang eksponensial.

Adapun kompleksitas ruang dari pencarian *Iterative Depth Search* adalah $O(bd)$. Namun, berdasarkan data, terlihat bahwa dalam pencarian satu solusi, jumlah artikel yang dikunjungi pada kedalaman 3 lebih sedikit daripada kedalaman 2. Hal ini dikarenakan proses

pencarian akan berhenti apabila telah ditemukan satu solusi saja. Metode solusi banyak merupakan metode yang paling tepat untuk menggambarkan kompleksitas ruang dari *Iterative Depth Search* karena pada dasarnya notasi *Big-O* memang menyatakan kemungkinan terburuk, yaitu mengunjungi setiap *node* yang ada. Notasi *Big-O* ini menggambarkan tingkat peningkatan jumlah *node* yang dikunjungi dikarenakan pencarian akan berhenti jika telah memeriksa semua *node* pada kedalaman maksimum pencarian, memiliki suatu solusi dan telah melakukan iterasi untuk seluruh link yang ada pada depth tersebut.

4.4.3. Perbandingan Algoritma BFS dengan IDS

Berdasarkan hasil uji sebelumnya, terlihat bahwa pencarian dengan BFS memakan waktu lebih sedikit dibandingkan pencarian dengan IDS tanpa adanya *cache*, dan pencarian dengan IDS dengan adanya *cache* rata-rata lebih cepat dibandingkan pencarian dengan IDS.

Hal tersebut terjadi karena adanya perbedaan penerapan konsep yang digunakan untuk membantu mempercepat proses pencarian. Pada algoritma pencarian dengan BFS, digunakan konsep *multithreading* untuk proses *scraping* link dari artikel-artikel pada kedalaman yang sama karena sifat proses pencarian BFS memungkinkan konsep tersebut. Sedangkan pada pencarian IDS, konsep *multithreading* tidak dapat digunakan karena pencarian harus masuk terlebih dahulu ke link pada kedalaman $d+1$ sebelum melanjutkan ke link lain pada kedalaman d .

Konsep *multithreading* membuat penambahan *node* pada kedalaman $d+1$ bersifat acak dan tidak terurut. Hal ini tidak menjadi masalah pada algoritma BFS, karena yang terpenting dari pencarian BFS adalah menyelesaikan terlebih dahulu pemeriksaan *node-node* pada kedalaman d sebelum lanjut memeriksa pada kedalaman $d+1$. Sehingga jika urutan *node-node* pada kedalaman $d+1$ tidak sesuai dengan urutan *node-node* pada kedalaman d , proses pemeriksaan BFS masih terpenuhi. Hal ini berbanding terbalik dengan proses pencarian IDS, pemeriksaan *node* pada kedalaman d harus segera diikuti dengan pemeriksaan *node-node* tetangga pada kedalaman $d+1$ dari node tersebut. Sifat *multithreading* yang acak membuat aturan ini dilanggar.

Adapun penggunaan *cache*, setelah dilakukan uji coba sebelum-sebelumnya, ketika digabungkan dengan sifat *multithreading*, membuat solusi proses pencarian pada BFS menjadi tidak konsisten, dimana jumlah solusi yang dihasilkan senantiasa berubah-ubah setiap kali dilakukan pengulangan. Hal ini kemungkinan terjadi karena penggunaan *cache* membuat proses pencarian solusi menjadi terlalu cepat. Namun ketika ada suatu node yang belum ada di *cache*, akan dilakukan proses *scraping* link yang memakan waktu lebih lama daripada hanya dengan membaca *cache*. Proses *scraping* link yang lebih lama kemungkinan membuat hasil pembacaan *cache* terhapus ketika hasil *scraping* dimasukkan ke dalam *queue* akibat adanya *race condition*. Oleh karena itulah pada BFS tidak digunakan konsep *cache*. Adapun karena IDS tidak memanfaatkan *multithreading*, maka konsep *cache* jadi dapat diterapkan.

Oleh karena perbedaan konsep yang digunakan, maka tidak dapat dilakukan perbandingan efektivitas proses pencarian dengan BFS dan IDS.

BAB V

PENUTUP

5.1. Kesimpulan

Wikirace adalah sebuah permainan atau perlombaan untuk berpindah dari sebuah artikel wikipedia ke artikel wikipedia lainnya dengan melakukan klik pada artikel lain yang terdapat dalam sebuah artikel wikipedia hingga mencapai artikel wikipedia tujuan. Untuk mempermudah pencarian ini, dibuatlah sebuah Wikirace Pathfinder yang dapat mencari *path* dari sebuah artikel awal hingga artikel tujuan. Wikirace Pathfinder ini diwujudkan dengan dua tipe algoritma *searching*, yaitu *Breadth First Search* dan *Iterative Deepening Search*.

Berdasarkan hasil uji coba yang dilakukan, dapat dilihat bahwa Algoritma BFS dan IDS memiliki perilaku yang berbeda. Algoritma BFS akan memiliki waktu yang relatif lebih lama dibandingkan IDS dalam pencarian satu solusi, akan tetapi memiliki waktu yang relatif lebih cepat dibandingkan IDS dalam pencarian banyak solusi. Konsep *multithreading* dapat diterapkan pada algoritma BFS karena sifat pencarian BFS yang terlebih dahulu menyelesaikan pencarian pada kedalaman d sebelum berpindah ke kedalaman $d+1$. Sedangkan pada algoritma IDS, konsep *multithreading* tidak dapat dilakukan karena algoritma IDS harus memasuki *node* secara mendalam sebelum berpindah ke *node* di sebelahnya, sehingga urutan pemeriksaan *node* menjadi penting. Karena pemeriksaan yang berulang-ulang pada algoritma IDS, maka konsep *cache* dapat diterapkan pada algoritma ini.

Karena perbedaan konsep yang digunakan antara algoritma BFS dengan IDS, maka tidak dapat dilakukan perbandingan secara langsung untuk melihat keefektifan kedua algoritma.

5.2. Saran

Beberapa saran dari kelompok kami di antaranya:

1. Lebih ditingkatkan lagi efektivitas, efisiensi, serta *readability* dari algoritma yang telah dibuat.
2. Usahakan penggerjaan tugas besar lebih terarah dan teratur kedepannya.

5.3. Refleksi

Selama penggeraan tugas besar ini kami mendapatkan beberapa pelajaran. Selain dari semakin meningkatnya pemahaman mengenai algoritma traversal graf IDS dan BFS, kami juga semakin memahami mengenai *web development front-end* dan *back-end*. Selain itu, kami juga mengetahui bahwa sangat sulit untuk mendapatkan waktu yang cepat (kurang dari 5 menit) dalam penyelesaian permainan ini. Secara keseluruhan kami mendapatkan *insight* dalam banyak hal.

LAMPIRAN

Repository Github

Link repository :

- *Front-End* : https://github.com/Agil0975/Tubes2_FE_Penyelam_Handal.git
- *Back-End* : https://github.com/raflyhangga/Tubes2_BE_Penyelam_Handal.git

Video

Link YouTube : <https://youtu.be/6TqHLVBLFIM>

DAFTAR PUSTAKA

IF2211 Strategi Algoritma - Semester II Tahun 2023/2024. (n.d.). Homepage Rinaldi Munir.

Retrieved April 27, 2024, from

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/stima23-24.htm>