LAPORAN TUGAS BESAR II
IF2211 STRATEGI ALGORITMA

PEMANFAATAN ALGORITMA BFS DAN IDS DALAM PERMAINAN WIKIRACE

Kelompok BarengApin

Disusun oleh:

13522019	Wilson Yusda
13522045	Elbert Chailes
13522081	Albert

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG 2023

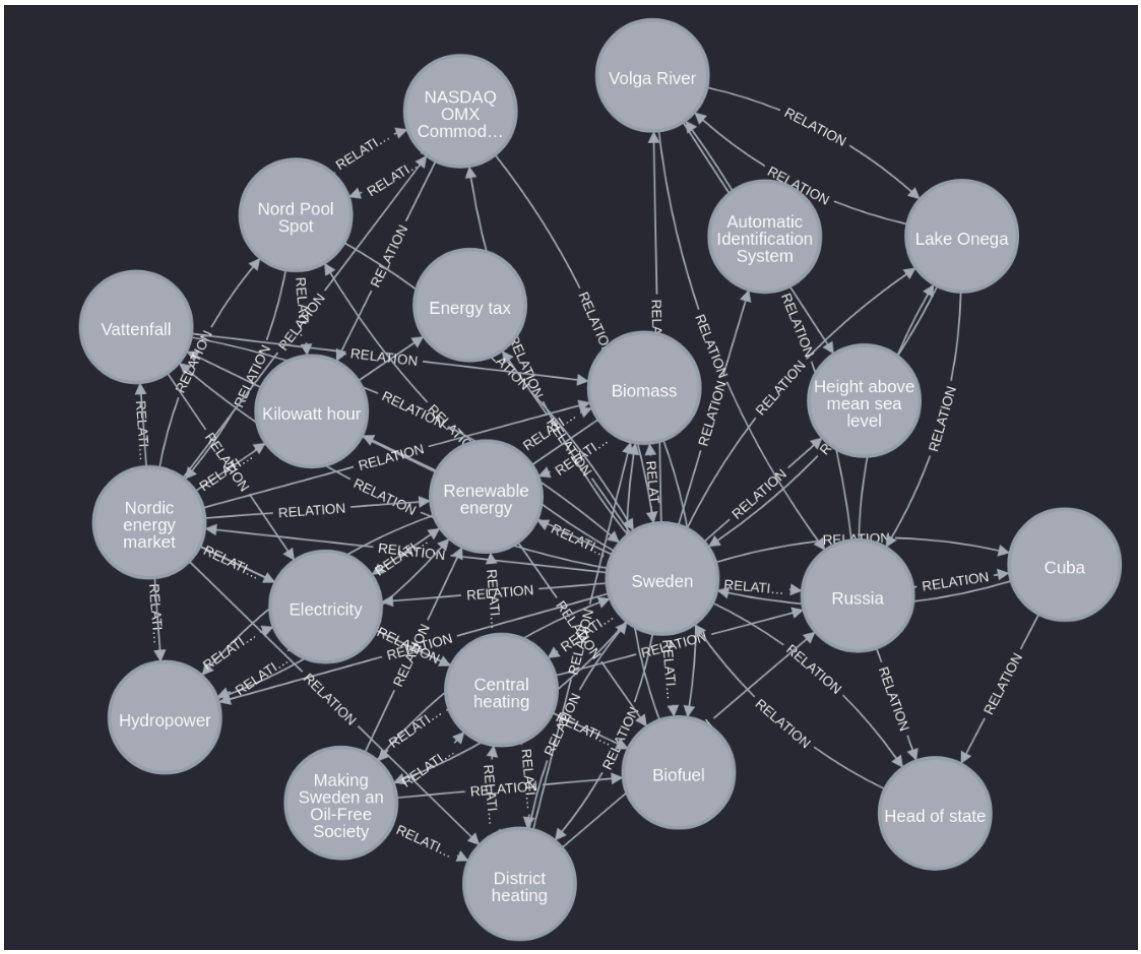
DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB 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.



Gambar 1.1. Ilustrasi Graf WikiRace (Sumber: https://miro.medium.com/v2/resize:fit:1400/1*jxmEbVn2FFWybZsIicJCWQ.png)

BAB II
LANDASAN TEORI

2.1. Penjelajahan Graf

Penjelajahan graf, sering disebut juga *graph traversal*, merupakan sebuah proses mengunjungi simpul-simpul yang terdapat dalam sebuah graf dengan cara yang sistematik. Pengunjungan yang dilakukan memiliki syarat setiap simpul atau *node* yang terdapat pada graf hanya dapat dikunjungi tepat sekali. Penjelajahan ini digunakan untuk menyelesaikan berbagai macam permasalahan komputasi dan analisis data, seperti mencari jalur terpendek, mendeteksi siklus, atau menjelajahi semua simpul yang terhubung dalam jaringan.

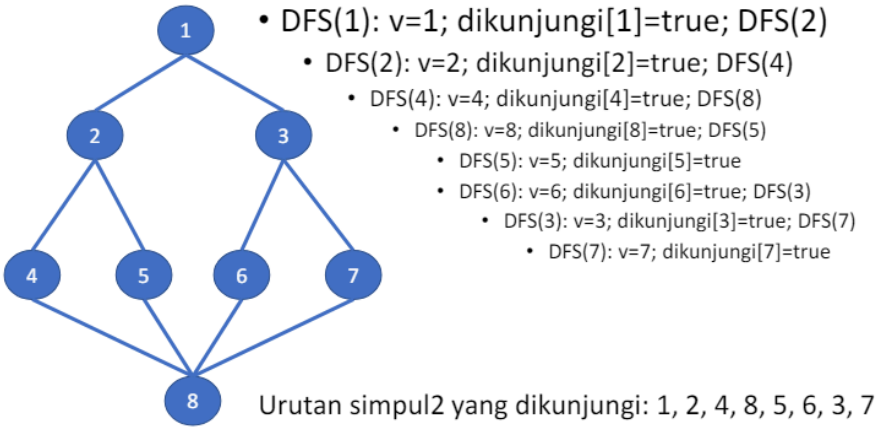
Ada dua pendekatan umum yang paling sering digunakan untuk melakukan penjelajahan graf, yaitu pencarian dengan *depth-first search* (DFS, penelusuran dalam pertama) dan *breadth-first search* (BFS, penelusuran bebas pertama).

2.2. Penelusuran Dalam Pertama (*Depth-First Search* / DFS)

Algoritma penelusuran dalam pertama atau *Depth-First Search* (DFS) adalah algoritma penelusuran atau pencarian yang digunakan untuk menjelajahi semua simpul dalam struktur graf dari titik awal tertentu. DFS melakukan eksplorasi setiap cabang dari graf sejauh mungkin sebelum mundur ke simpul sebelumnya dan mencoba jalur alternatif lainnya.

Secara umum, algoritma DFS dapat digambarkan dengan studi kasus berikut. Misalkan terdapat traversal dimulai dari simpul v. Maka, algoritma akan berjalan sesuai berikut.

1. Kunjungi simpul v
2. Kunjungi simpul w yang bertetangga dengan simpul v.
3. Ulangi DFS mulai dari simpul w.
4. Ketika mencapai simpul u sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian runut-balik (*backtrack*) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul w yang belum dikunjungi.
5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi



Gambar 2.2. **Ilustrasi Penjelajahan Graf DFS**

2.3. Penelusuran Bebas Pertama (*Breadth-First Search* / BFS)

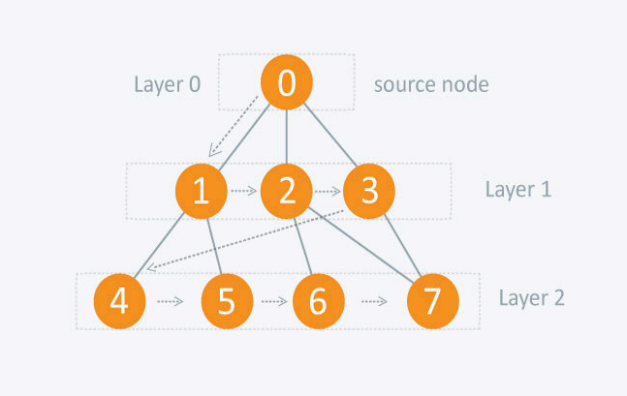
Algoritma *Breadth-First Search* (BFS) adalah salah satu algoritma pencarian yang digunakan untuk menjelajahi atau mencari jalur terpendek dari satu simpul ke simpul lain dalam sebuah graf yang tidak berbobot. Cara kerja BFS adalah dengan mengeksplorasi semua simpul yang bersebelahan dengan simpul saat ini sebelum melanjutkan ke simpul-simpul yang lebih jauh. Berikut adalah penjelasan lebih detail tentang cara kerja dan keunggulan BFS:

1. Pemilihan simpul berikutnya secara melebar. Saat menjalankan BFS, algoritma akan memilih simpul berikutnya untuk dieksplorasi berdasarkan prinsip melebar. Artinya, algoritma akan mengeksplorasi semua simpul yang bersebelahan dengan simpul saat ini terlebih dahulu sebelum melanjutkan ke simpul-simpul yang lebih jauh.
2. Penggunaan struktur data queue. Untuk mengatur urutan pemrosesan simpul, BFS menggunakan struktur data queue. Setiap kali sebuah simpul diproses, simpul-simpul yang bersebelahan dengan simpul tersebut akan dimasukkan ke dalam queue. Simpul yang akan diproses selanjutnya adalah simpul yang berada di depan queue, sesuai dengan prinsip *First In First Out* (FIFO) dari struktur data queue.
3. Jalur terpendek. Salah satu keunggulan utama BFS adalah kemampuannya untuk menemukan jalur terpendek dalam graf yang tidak berbobot. Dengan mengeksplorasi

simpul-simpul secara melebar, BFS akan menemukan jalur terpendek dari simpul awal ke simpul tujuan, asalkan jalur tersebut ada.

4. Kondisi berhenti. Algoritma BFS akan terus berjalan sampai semua simpul sudah dikunjungi atau simpul tujuan sudah ditemukan. Ketika semua simpul sudah dikunjungi atau simpul tujuan sudah ditemukan, algoritma akan berhenti.

BFS menjadi salah satu algoritma pencarian yang penting dan sering digunakan dalam berbagai aplikasi, seperti perutean jaringan komputer, pencarian jalur terpendek dalam peta, dan lain sebagainya.



Gambar 2.3. Ilustrasi BFS

2.4. Aplikasi Website

Go (atau Golang) adalah bahasa pemrograman yang dikembangkan oleh Google. Go dirancang untuk memudahkan pengembangan perangkat lunak yang sederhana, efisien, dan skalabel. Go juga dikenal karena kompilasinya yang cepat dan kemampuannya dalam menangani aplikasi berskala besar. Dalam aplikasi website, Go sering digunakan untuk membuat backend server. Go memiliki library standar yang kaya, termasuk untuk menangani HTTP, *parsing* JSON, dan lain-lain, yang membuatnya ideal untuk pengembangan web.

Dalam proyek ini, web scraping memainkan peran penting dalam mengumpulkan data dari berbagai sumber di web. Web scraping adalah proses ekstraksi informasi dari halaman web secara otomatis menggunakan bot atau web crawler. Dengan web scraping, kita dapat mengumpulkan data seperti harga produk atau informasi lainnya yang dapat

digunakan untuk analisis atau keperluan lainnya. Go memiliki beberapa library yang dapat digunakan untuk web scraping, salah satunya adalah Go Colly. Go Colly adalah sebuah library web scraping yang dibuat khusus untuk bahasa pemrograman Go. Go Colly menyediakan fitur-fitur yang memudahkan pengembang dalam membuat web scraper, seperti pengaturan kecepatan scraping, ekstraksi data dengan selector CSS, dan penanganan error dan retry.

Next.js, di sisi lain, adalah framework React yang populer untuk pengembangan aplikasi web front-end. Next.js memperluas fitur React dengan menyediakan fungsionalitas tambahan seperti rendering server-side, static site generation, dan routing yang lebih kuat. Kombinasi antara Go untuk backend dan Next.js untuk frontend dapat memberikan pengalaman yang lebih baik dalam pembuatan aplikasi website. Go dapat digunakan untuk menangani logika dan layanan API, sementara Next.js dapat digunakan untuk membuat antarmuka pengguna yang dinamis dan interaktif.

BAB III
ANALISIS PEMECAHAN MASALAH

3.1. Langkah-langkah Pemecahan Masalah

Berikut adalah langkah-langkah pemecahan masalah dengan algoritma BFS *single-path*.

1. Algoritma dimulai dengan menerima URL awal dari frontend. URL ini dianggap sebagai *node* awal dan titik start pencarian.
2. *Node* URL awal akan discrap untuk mendapatkan anak-anak URL yang berhubungan. Anak-anak URL ini adalah *node-node* yang akan dieksplorasi lebih lanjut dan ditempatkan dalam antrian (queue) pencarian.
3. Setiap *node* URL yang diperoleh dari hasil *scraping* dievaluasi. Jika URL tersebut tidak sesuai dengan target URL yang diinginkan, maka URL tersebut akan dimasukkan kembali ke dalam queue untuk dieksplorasi pada kedalaman yang lebih dalam.
4. Proses pencarian akan terus berlanjut, mengecek setiap URL yang di-scrap apakah cocok dengan URL target. Jika ditemukan URL yang cocok, algoritma akan berhenti dan mengembalikan URL tersebut sebagai hasil pencarian.
5. Jika URL target tidak ditemukan dalam iterasi pertama, queue yang dihasilkan dari langkah sebelumnya akan diulangi untuk scrapping dan pengecekan lebih lanjut.
6. Langkah-langkah *scraping* dan pengecekan akan diulang terus-menerus sampai target ditemukan atau semua kemungkinan telah dieksplorasi.

Berikut adalah langkah-langkah pemecahan masalah dengan algoritma BFS *multi-path*.

1. Algoritma dimulai dengan menerima URL awal dari frontend. URL ini dianggap sebagai *node* awal dan titik start pencarian.
2. *Node-node* yang diperoleh dimasukkan ke dalam queue, ini dilakukan untuk mempersiapkan sebuah struktur data untuk menyimpan semua jalur yang valid hingga ke target.
3. Setiap *node* discrapping kemudian dievaluasi. Jika sebuah *node* cocok dengan target pada kedalaman tertentu, jalur ke *node* tersebut disimpan sebagai hasil. Namun, pencarian tidak berhenti dan terus melanjutkan ke *node* lain di kedalaman yang sama untuk memastikan semua jalur potensial ditemukan.

4. Setelah satu kedalaman penuh dieksplorasi, algoritma hanya akan melanjutkan ke kedalaman berikutnya jika belum ada jalur yang ditemukan hingga target
5. Proses ini diulang, meliputi scrapping lanjutan, evaluasi, dan penyimpanan jalur hingga semua kemungkinan jalur di kedalaman target telah dieksplorasi atau semua *node* telah ditinjau.

BFS *single-path* fokus pada penemuan jalur pertama yang valid dari titik awal ke target. Algoritma ini menghentikan pencarian segera setelah menemukan jalur pertama yang sesuai, sehingga efisien ketika hanya satu solusi yang diperlukan. Ini mengikuti proses dimana setiap URL yang di-*scrap* dievaluasi dan, jika tidak sesuai, dikembalikan ke dalam queue untuk dieksplorasi lebih lanjut, menghentikan pencarian seketika setelah menemukan cocokan dengan URL target. Sebaliknya, BFS multi-path bertujuan untuk mengidentifikasi semua jalur yang mungkin dari titik awal ke target yang berada pada kedalaman yang sama. Proses ini tidak berhenti setelah menemukan jalur pertama tetapi terus mengeksplorasi semua *node* di kedalaman di mana jalur pertama ditemukan untuk memastikan semua jalur potensial menjadi solusi yang diinginkan. Ini sangat berguna untuk analisis yang memerlukan pemahaman menyeluruh tentang semua rute yang mungkin antara dua titik.

Perbedaan utama antara BFS *single-path* dan *multi-path* terletak pada penanganan hasil setelah jalur pertama ditemukan. *Single-path* berhenti segera setelah jalur ditemukan, membuatnya lebih cepat dalam skenario di mana hanya satu hasil yang dibutuhkan. *Multi-path*, di sisi lain, melanjutkan pencarian untuk mengumpulkan semua jalur yang memungkinkan pada kedalaman yang sama, memberikan pandangan yang lebih komprehensif tetapi membutuhkan lebih banyak sumber daya dan waktu untuk menjalankan karena ia mengeksplorasi lebih banyak *node*.

Berikut adalah langkah-langkah pemecahan masalah dengan algoritma IDS *single-path*.

1. Algoritma dimulai dengan menerima URL awal dimulainya pencarian sebagai titik awal pencarian (*starting node*) dan URL akhir yang dituju yang kemudian dianggap sebagai titik akhir *node* yang dituju (*goal node*).
2. URL yang menjadi *starting node* akan di-*scrap* dan membangkitkan anak-anak dari *starting node* berupa kumpulan link yang terdapat pada *parent node* tersebut. Pembangkitan sebanyak satu kali terhadap *parent node* berarti terjadi penambahan depth sebanyak satu pula.
3. Pada proses pembangkitan *node-node* anak dari *parent node*, maka algoritma akan melakukan pengecekan apakah *node* anak yang dibangkitkan mengandung *url* yang sedang dituju, dengan kata lain mengecek apakah *node* anak merupakan *goal node* atau bukan. Maka, terdapat dua kemungkinan yang dapat terjadi.
    a. Jika *node* anak merupakan *goal node*, maka algoritma akan berhenti dan algoritma tidak akan melanjutkan pencarian terhadap *node* anak lainnya.
    b. Jika *node* anak BUKAN merupakan *goal node*, maka algoritma akan melanjutkan pencarian terhadap *node-node* anak yang sederajat.
4. Pembangkitan anak (langkah 2 dan 3) akan dilakukan secara terus menerus dengan *depth* yang telah dibatasi, misalnya kedalaman n (DLS dengan *depth* n).
5. IDS melakukan pembangkitan anak secara berulang, dengan kedalaman yang bertambah secara progresif pula, dengan kata lain IDS akan melakukan pencarian hingga mendapatkan solusi dengan melakukan DLS secara berulang dengan melakukan inkremen kedalaman n. Misal, n mulai dari *depth 0..*x (misal x merupakan kedalaman dimana DLS mendapatkan solusi pertama) dan x merupakan akan terus bertambah dari 0 hingga sebuah kedalaman dimana algoritma DLS ini mendapatkan solusi pertamanya.

Berikut adalah langkah-langkah pemecahan masalah dengan algoritma IDS *multi-path*.

1. Algoritma dimulai dengan menerima URL awal dimulainya pencarian sebagai titik awal pencarian (*starting node*) dan URL akhir yang dituju yang kemudian dianggap sebagai titik akhir *node* yang dituju (*goal node*).
2. URL yang menjadi *starting node* akan di-*scrap* dan membangkitkan anak-anak dari *starting node* berupa kumpulan link yang terdapat pada *parent node* tersebut.

Pembangkitan sebanyak satu kali terhadap *parent node* berarti terjadi penambahan depth sebanyak satu pula.

3. Pada proses pembangkitan *node-node* anak dari *parent node*, maka algoritma akan melakukan pengecekan apakah *node* anak yang dibangkitkan mengandung *url* yang sedang dituju, dengan kata lain mengecek apakah *node* anak merupakan *goal node* atau bukan. Maka, terdapat dua kemungkinan yang dapat terjadi.
    a. Jika *node* anak merupakan *goal node*, maka algoritma akan terus berjalan hanya dan melakukan pengecekan seluruh *node* anak yang telah terlanjur dibangkitkan atau semua *node* anak yang merupakan saudara dengan *goal node*. Oleh karena itu, solusi menjadi lebih dari satu.
    b. Jika node anak BUKAN merupakan *goal node*, maka algoritma akan melanjutkan pencarian terhadap *node-node* anak yang sederajat.
4. Pembangkitan anak (langkah 2 dan 3) akan dilakukan secara terus menerus dengan *depth* yang telah dibatasi, misalnya kedalaman n (DLS dengan *depth* n).
5. IDS melakukan pembangkitan anak secara berulang, dengan kedalaman yang bertambah secara progresif pula, dengan kata lain IDS akan melakukan pencarian hingga mendapatkan solusi dengan melakukan DLS secara berulang dengan melakukan inkremen kedalaman n. Misal, n mulai dari *depth 0..*x (misal x merupakan kedalaman dimana DLS mendapatkan solusi pertama) dan x merupakan akan terus bertambah dari 0 hingga sebuah kedalaman dimana algoritma DLS ini mendapatkan solusi pertamanya.

Perbedaan yang terdapat antara langkah pemecahan masalah IDS *single-path* dengan IDS *multi-path* adalah hanya pada langkah ketiga, yang mana jika pada kasus IDS *single-path* dan bertemu kasus pengecekan *node* anak merupakan *goal node*, maka algoritma IDS akan berhenti pada saat itu juga dan mengembalikan hasil perjalanan tempuh. Sedangkan, untuk IDS *multi-path* ketika bertemu kasus tersebut, algoritma ini cenderung untuk melanjutkan pencarian *node* anak lainnya yang bersaudara dengan *goal node*, atau dengan kata lain mencari posibilitas solusi lain selain solusi pertama yang didapatkan. Namun, pencarian kemungkinan solusi lain tidak dilanjutkan pada kedalaman selanjutnya, dimana ditandai dengan kedua IDS yang hanya akan melakukan DLS hingga kedalaman x (dengan x merupakan kedalaman dimana solusi ditemukan pertama kali.

Jadi, hasil solusi dari IDS *multi-path* meskipun memiliki hasil yang lebih dari satu, namun jarak yang ditempuh pada hasil akan memiliki ukuran yang sama semuanya karena dicari pada kedalaman yang sama pula.

3.2. Proses Pemetaan Masalah

Berikut pemetaan persoalan pada BFS single path dan BFS multi path.

1. Setiap URL direpresentasikan sebagai *node* dalam graf. Untuk menghindari pengulangan dan loop yang tidak perlu, atribut 'visited' digunakan. Atribut ini menandai *node* mana saja yang telah dikunjungi, memastikan bahwa setiap *node* hanya diproses sekali. Node yang telah dikunjungi tidak akan ditambahkan kembali ke dalam queue untuk dieksplorasi pada kedalaman yang lebih dalam, kecuali jika diperlukan dalam konteks BFS *multi-path* untuk mengeksplorasi semua jalur pada kedalaman tertentu.
2. *Queue* digunakan untuk mengatur *node-node* yang harus diproses pada kedalaman selanjutnya. Dalam BFS, penting untuk menyimpan relasi *parent-child* agar dapat merekonstruksi jalur kembali ke *node* asal. Untuk setiap *node* yang diproses, informasi tentang *node parent*-nya disimpan. Ini memungkinkan algoritma untuk melacak jalur lengkap dari *node* awal ke *node* target saat solusi ditemukan.
3. Hasil yang ditemukan, baik dalam BFS *single-path* maupun *multi-path*, disimpan dalam *array* dinamis. Tiap solusi yang berhasil memenuhi kriteria pencarian (mencapai target URL) dicatat dalam *array* ini. Untuk BFS *single-path*, pencarian berhenti setelah jalur pertama ditemukan dan disimpan. Sementara untuk BFS *multi-path*, semua jalur yang ditemukan pada kedalaman yang sama dengan jalur pertama yang ditemukan juga akan disimpan dalam array ini, memberikan gambaran komprehensif tentang semua rute yang memungkinkan.

Berikut pemetaan persoalan pada IDS *single-path* dan IDS *multi-path*.

1. Setiap url akan direpresentasikan sebagai *node* dari graf. Atribut *visited* akan digunakan untuk menandai *node-node* yang sudah pernah dikunjungi oleh algoritma. Atribut ini digunakan untuk mengingatkan kepada algoritma agar tidak mengunjungi sebuah *url* lebih dari sekali dengan harapan bahwa algoritma tidak membuang waktu untuk kembali ke *url* yang memang sudah tidak memiliki solusi. Oleh karena itu, dengan dilakukannya

pengecekan atribut *visited*, jika *node* url sudah pernah dikunjungi maka tidak akan dimasukkan pada atribut tersebut, dan sebaliknya jika *node* url belum pernah ditempuh oleh algoritma maka akan dimasukkan ke atribut *visited* dan dilakukan proses pencarian terhadap *node* url jika solusi masih belum ditemukan.

2. *Parent-parent node* yang ditempuh oleh algoritma disimpan dalam sebuah atribut berupa *path*. Atribut *path* menyimpan *node-node* yang dilalui oleh algoritma sehingga pada setiap penambahan kedalaman, *node* yang terdapat pada *path* akan bertambah banyak sesuai dengan kedalaman yang ditempuh oleh algoritma. Tujuan adanya atribut ini adalah dengan alasan untuk menyimpan informasi perjalanan yang telah ditempuh oleh algoritma pada setiap pencarian yang dilakukan baik *forward* maupun ketika algoritma melakukan *backtracking*. Atribut inilah yang pada akhirnya akan dikembalikan jika solusi *goal node* telah ditemukan pada *node* anak.
3. Atribut *depth* yang digunakan untuk melakukan penyimpanan informasi atau *tracking* terhadap kedalaman yang telah ditempuh oleh algoritma sehingga akan sesuai dengan *limit* yang akan dicapai oleh algoritma tersebut.
4. Terdapat juga atribut *cache* yang bertujuan untuk menyimpan hasil *scraping* yang dilakukan pada setiap *node* yang sudah pernah ditempuh sehingga tidak perlu melakukan perlakuan scraping secara repetitif terhadap url yang sudah pernah ditempuh. Dengan itu, maka akan terjadi penghematan waktu dengan melakukan pengambilan data dari hasil *caching* yang disimpan pada atribut tersebut dibandingkan dengan melakukan *scraping* secara repetitif.

3.3 Fitur Fungsional dan Arsitektur *Website*

3.3.1. Arsitektur *Website*

Dalam pembangunan sebuah website, rancangan dan fondasi utama dalam pembangunannya adalah *frontend* (yang mengatur tampilan / *client-sided*) dan *backend* (yang mengatur pengaliran data yang tidak terlihat oleh klien / *server-sided*). Maka dari itu, dalam folder root utama atau *src* memiliki *folder tree / structure* seperti berikut.

```
├──backend
│       bfs.go
│       go.mod
│       go.sum
```

```
│       ids.go
│       main.go
│       multi-ids.go
│       utils.go
│
└──frontend
    │   .eslintrc.json
    │   .gitignore
    │   next.config.mjs
    │   package-lock.json
    │   package.json
    │   postcss.config.mjs
    │   README.md
    │   tailwind.config.ts
    │   tsconfig.json
    │
    ├──app
    │   │   favicon.ico
    │   │   globals.css
    │   │   layout.tsx
    │   │   page.tsx
    │   │
    │   ├──About-Us
    │   │       page.tsx
    │   │
    │   ├──component
    │   │       Navbar.tsx
    │   │       SearchInput.tsx
    │   │       Loading.tsx
    │   │
    │   └──LandingPage
    │           page.tsx
    │
    └──public
            Bumi.png
            github.png
            instagram.png
            next.svg
            Nezuko.gif
            Son.jpg
            swap.png
            vercel.svg
            wiki.png
            no-image.png
```

Untuk pembangunan *website* ini, kelompok BarengApin menggunakan teknologi frontend *NextJS* (yang berbasis bahasa pemrograman *typescript*) yang dipadukan bersama dengan teknologi backend *gin* (yang berbasis bahasa pemrograman *go*). Selain itu, untuk membantu proses pembangunan dan realisasi UI (*user-interface*) *website*, kelompok BarengApin menggunakan teknologi *TailwindCSS* yang memudahkan secara penulisan

untuk mempermudah dan mempercepat pembangunan website secara UI. Dalam *repository* yang menyimpan semua kebutuhan yang dibutuhkan untuk membangun *website* ini, dibagi menjadi dua folder utama, yaitu frontend dan backend dengan tujuan modularitas dan memisahkan secara jelas batasan frontend dan backend.

Dalam *folder backend* memiliki struktur file yang cukup modular dengan perlakuan pemisahan file antara API dengan fungsi-fungsi yang menampung algoritma utama dalam tugas besar kali ini, yaitu IDS dan BFS. Struktur file yang terdapat pada folder backend adalah sebagai berikut.

1. **bfs.go** merupakan *file* yang berisi fungsi-fungsi algoritma BFS dan algoritma Multi-BFS, serta fungsi-fungsi pembantu lainnya untuk membantu keberjalanan algoritma fungsi BFS.
2. **go.mod** merupakan *file* yang digunakan untuk mendefinisikan path modul dan dependensi proyek Go,
3. **go.sum** berisi *checksum* kriptografis dari isi versi modul yang spesifik untuk memastikan integritas unduhan dan deteksi perubahan yang tidak sah.
4. **ids.go** merupakan *file* yang berisi fungsi-fungsi algoritma IDS yang digunakan untuk menyelesaikan persoalan dan fungsi-fungsi pembantu (seperti DLS, definisi tipe *caching* dan mekanismenya) untuk algoritma IDS.
5. **main.go** merupakan file utama atau otak dari folder *backend* karena berisi API dengan *route* / yang menyambungkan antara frontend dan backend, dan menggunakan fungsi yang berasal dari file *bfs.go*, *ids.go*, dan *multi-ids.go*.
6. **multi-ids.go** merupakan file yang berisi algoritma IDS untuk mencari solusi yang lebih dari satu, namun dengan kedalaman yang sama. Alasan dipisahkan dengan algoritma IDS utama yang berada pada file *ids.go* karena keduanya memiliki kemiripan yang cukup ekstrim sehingga lebih baik dipisah agar terjadi penekanan yang lebih jelas bahwa kedua fungsi tersebut berbeda antara IDS dan MultiIDS.
7. **utils.go** merupakan file yang berisi fungsi-fungsi pembantu untuk melakukan *scraping website* untuk membantu keberhasilan fungsi algoritma utama IDS dan BFS.

   Selain itu, juga terdapat folder frontend yang memiliki struktur file seperti

berikut.

1. ***Folder src/public/*** merupakan folder yang berisi aset-aset yang dapat diakses secara publik seperti gambar berbentuk png, jpg, ataupun berbentuk gif.
2. ***Folder src/app/*** merupakan folder utama dalam frontend ini karena mengandung kode pemrograman yang membangun *UI website*.
    a. ***Folder About-us/***, ***LandingPage/*** berisi file yang menampung informasi terkait masing-masing referensinya. Pada *website* ini tidak menggunakan *routing* sehingga pemisahan file hanya bertujuan untuk menandakan bahwa kedua *section* tersebut akan dipasang *reference* dengan tujuan *anchor scroll*.
    b. ***Folder Component/*** merupakan *folder* yang menampung komponen-komponen kecil yang mungkin dapat digunakan secara berulang pada file yang lain.
    c. ***File page.tsx*** merupakan otak utama dari *frontend* yang telah dibangun. Semua fungsi yang dibuat pada *file* lain pada akhirnya dipanggil dan digabungkan menjadi satu di *file* ini, dan *file* inilah yang pada akhirnya ditampilkan kepada pengguna *website*.
3. ***tailwind.config.ts*** adalah file konfigurasi *TypeScript* yang digunakan untuk mengatur pengaturan kompilasi dan proyek *TypeScript.*
4. ***tsconfig.json*** adalah file konfigurasi untuk *TailwindCSS* yang digunakan untuk menyesuaikan pengaturan seperti warna, ukuran, dan spasi dalam proyek *website*.

3.3.2. Implementasi Frontend

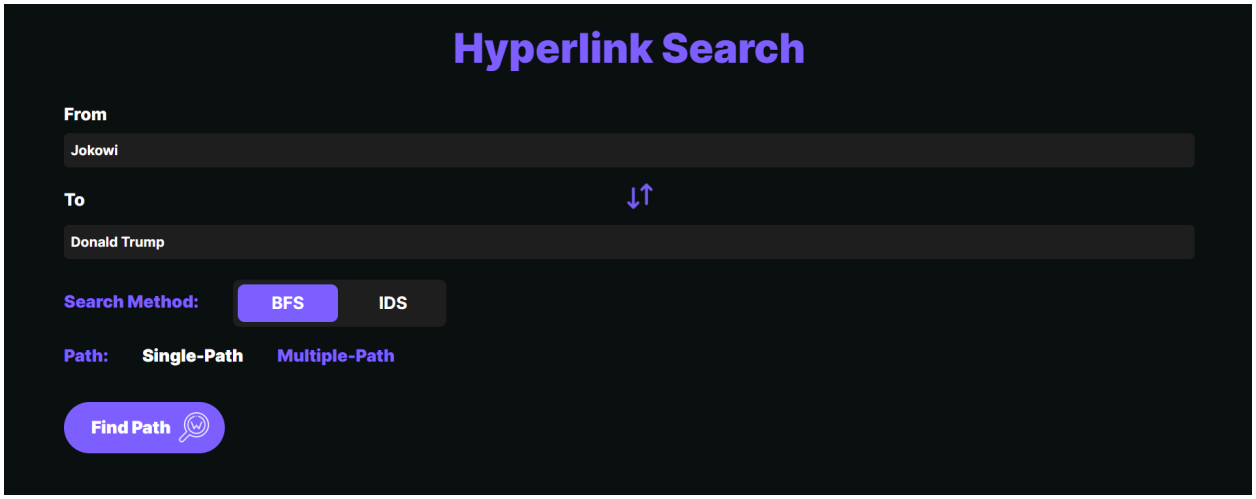
Implementasi fitur fungsional pada *website* ini secara *frontend* menggunakan *framework NextJS* yang didukung dengan *typescript* untuk pembuatan struktur laman serta *Tailwind CSS* untuk melakukan *styling* pada laman.

Implementasi dimulai dengan membuat komponen komponen penting yang nantinya akan digunakan dalam implementasi fitur fungsional. Komponen pertama yaitu *SearchInput* dimana komponen ini merupakan komponen yang mendukung fitur penambahan input dari pengguna serta memberikan informasi kepada *backend* untuk dilakukan pencarian jalur.

SearchInput dimulai dengan menerima kedua input pengguna, input tautan awal dan input tautan akhir. Sewaktu user memasukkan inputan, *Search Bar* akan secara otomatis memberikan sugesti atas tautan yang mungkin dimaksud pengguna. Hal ini dilakukan dengan melakukan pencocokkan dengan URL dari API wikipedia itu sendiri, dimana jumlah sugesti yang diberikan diberi batas sampai dengan 5 sugesti guna memastikan kenyamanan pengguna. Dalam pengembalian data dari URL, website hanya menyaring judul serta tautan dari sugesti dan akhirnya ditampilkan sebagai sugesti. Pengaksesan judul serta tautan dapat dilakukan dengan *s.title s.link* dengan s merupakan sugesti dengan nilai *title* dan *link*.

Komponen ini juga termasuk dalam pengiriman data ke *backend* dimana hal ini dilakukan setelah penekanan tombol pencarian. Dalam tampilan *Navbar,* aplikasi komponen ini dapat diakses pada bagian *services.* Berbagai fitur fungsional utama yang didukung oleh komponen ini, yakni:

1. Pencarian rute *hyperlink* dari tautan awal ke tautan tujuan dengan memanfaatkan konsep BFS dengan berfokus pada penghasilan 1 jalur (*single path).*
2. Pencarian rute *hyperlink* dari tautan awal ke tautan tujuan dengan memanfaatkan konsep IDS dengan berfokus pada penghasilan 1 jalur (*single path).*
3. Pencarian rute *hyperlink* dari tautan awal ke tautan tujuan dengan memanfaatkan konsep BFS untuk menghasilkan berbagai jalur menuju tautan tujuan (*multiple path).*
4. Pencarian rute *hyperlink* dari tautan awal ke tautan tujuan dengan memanfaatkan konsep IDS untuk menghasilkan berbagai jalur menuju tautan tujuan (*multiple path).*

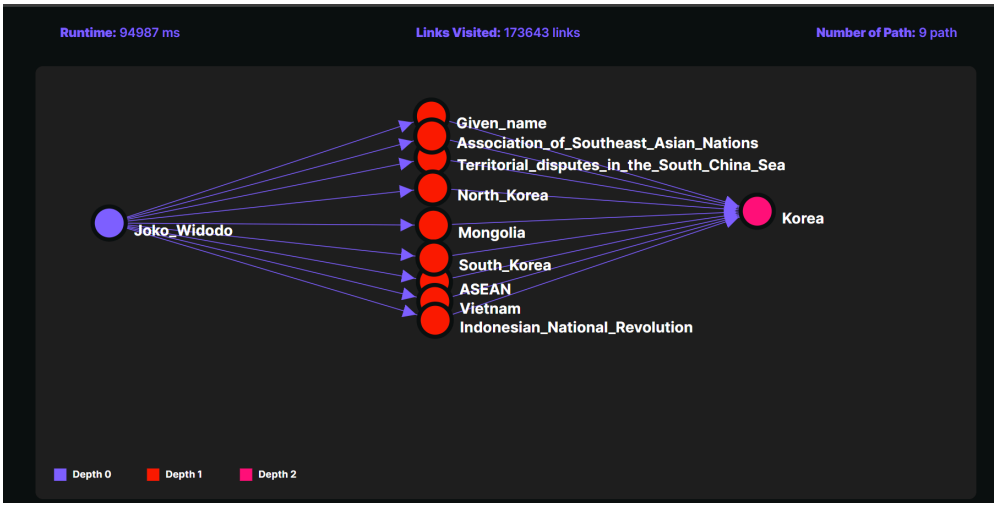


Gambar 3.3.2.1. Tampilan Fitur Pencarian *Website*

5. Penampilan rekomendasi untuk pengguna setelah mencocokkan masukkan dengan *Wikipedia*.

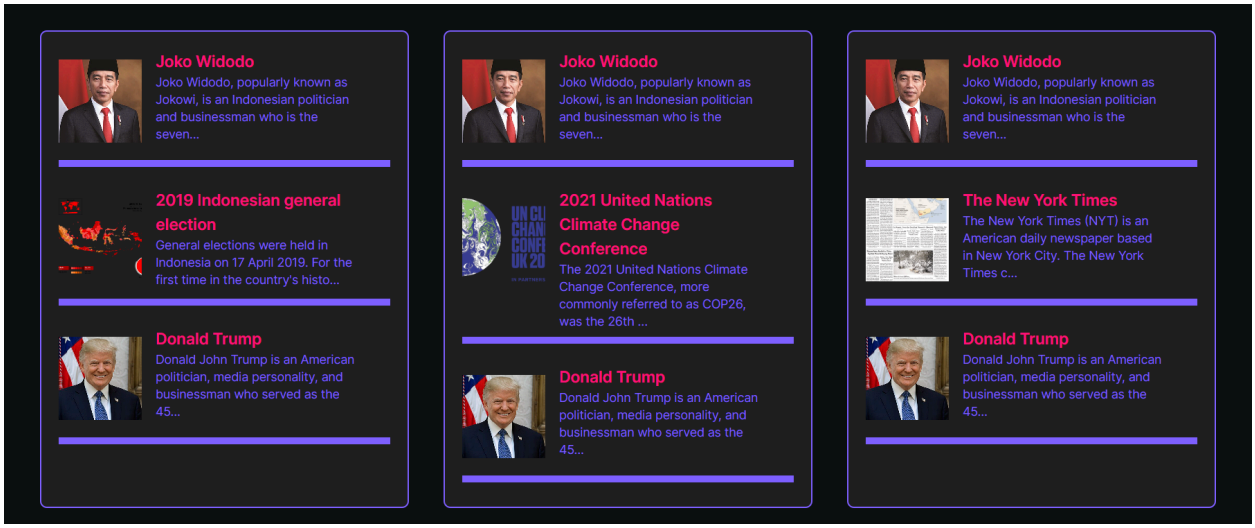
Gambar 3.3.2.2. Tampilan Fitur Rekomendasi *Website*

6. Penampilan jalur *hyperlink* dalam bentuk *Forced-Directed-Graph* yang didukung dengan pewarnaan untuk tiap kedalaman graf serta nama untuk setiap node.
7. Penampilan informasi terkait perjalanan algoritma seperti penampilan waktu pencarian, jumlah tautan yang dikunjungi, jumlah tautan yang dipilih, dan jumlah jalur yang dihasilkan.



Gambar 3.3.2.3. Tampilan Fitur Penampilan Hasil *Website*

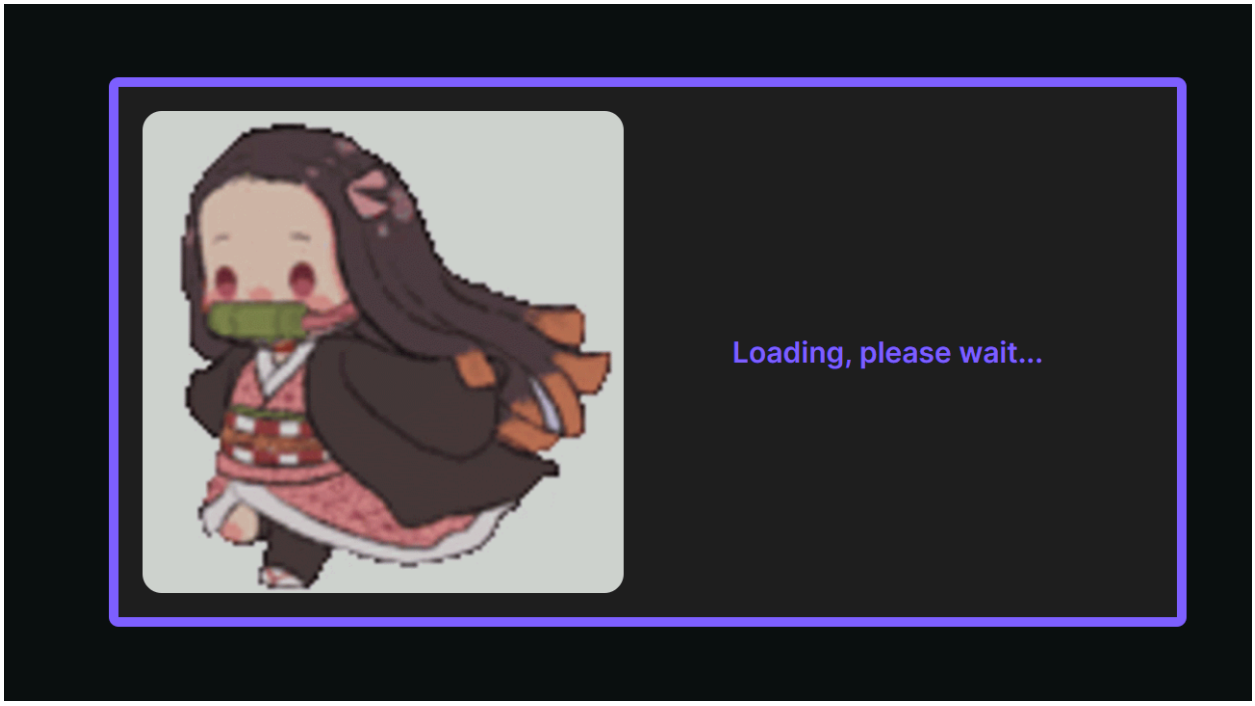
8. Pengecekan input tautan untuk memastikan kebenaran tautan yang dimasukkan.
9. Penampilan deskripsi singkat serta tampilan gambar untuk tiap node , tiap jalur yang dihasilkan dari pencarian.



Gambar 3.3.2.4. Tampilan Fitur Penampilan Detail Pencarian *Website*

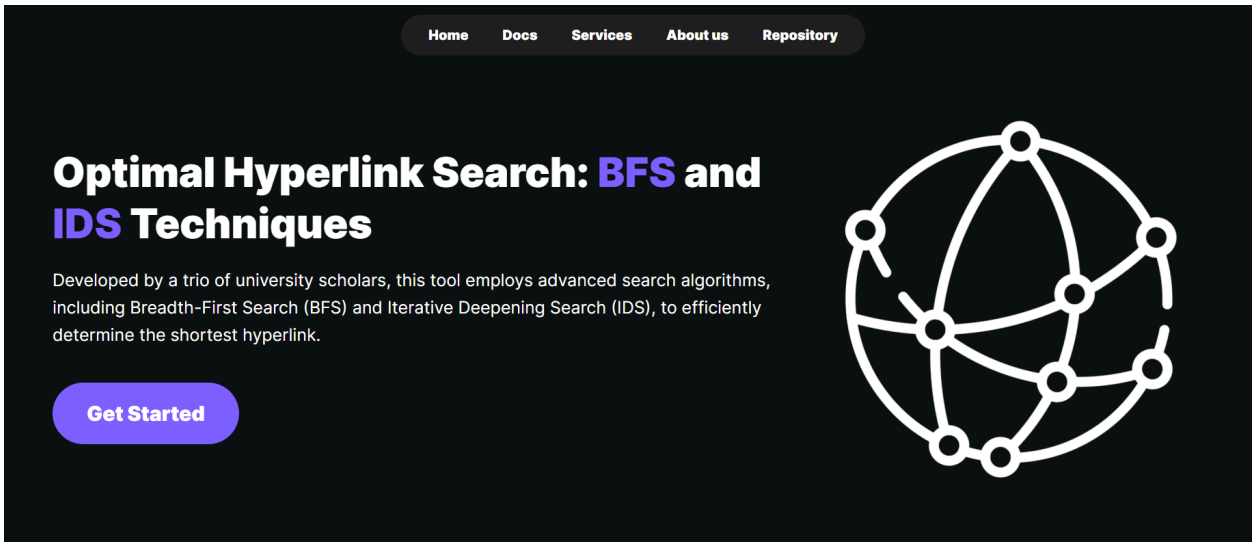
Adapun fitur tambahan untuk meningkatkan pengalaman dan kenyamanan pengguna dihasilkan dari beberapa komponen dan *page* pada folder *frontend*, yakni:

1. Komponen *Loading* untuk memberi tahu pengguna mengenai status pencarian serta memberi tahu apakah user memang memiliki spesifikasi yang sesuai dalam menjalankan program secara baik, baik dari sisi koneksi maupun penggunaan UI secara tepat.



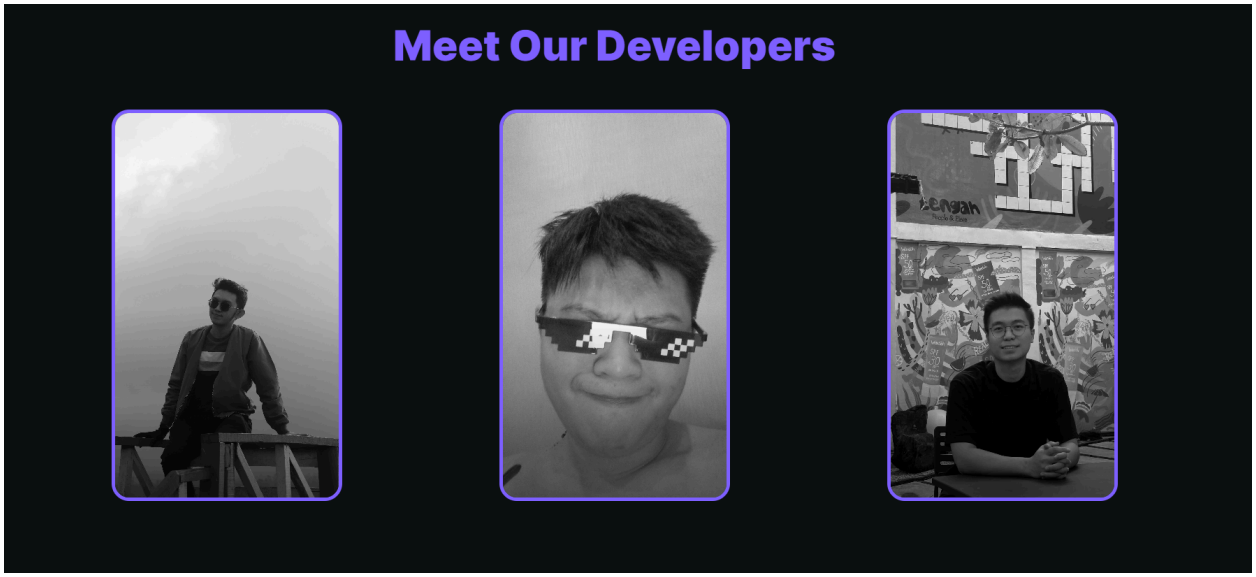
Gambar 3.3.2.5. Tampilan Fitur *Loading Website*

2. Komponen *Navbar* untuk memberikan gambaran struktur yang tersedia pada *website* serta sebagai *shortcut* untuk mengakses informasi yang dibutuhkan oleh pengguna.
3. Laman Utama sebagai wajah dari *website* ini untuk memberi gambaran kepada user mengenai tujuan dari pembuatan *website* ini.



Gambar 3.3.2.6. Tampilan Laman Utama *Website*

4. Laman *About Us* yang menampilkan semua *developer* dari pembuatan *website* ini sehingga mempermudah akses bagi pengguna untuk mencari informasi terkait pembuat *website*.



Gambar 3.3.26. Tampilan Laman *About Us Website*

3.3.3. Implementasi Backend

Implementasi pada *server-side* website (backend) menggunakan bahasa pemrograman *go* dan menggunakan *framework gin* dalam pembuatan API-nya. Sebagai informasi terkait *gin*, framework ini merupakan salah satu *framework go* yang kinerjanya minimalis dan sering digunakan untuk membuat REST API untuk backend.

Pada implementasi backend kali ini juga terdapat beberapa library yang digunakan, seperti *gin* tentunya karena menggunakan *framework* yang digunakan dan *go-colly* untuk membantu kegiatan *web-scraping* yang dilakukan pada *url-url* yang di-*visit*. API yang dibuat hanya tunggal dengan route / , yang telah meng-*handle request* "POST" method yang dikirimkan oleh frontend. Kemudian, API dengan *route* tersebut kemudian akan memberikan *response* yang berisi informasi *path* yang ditempuh oleh algoritma untuk mencapai *goal node*, waktu eksekusi algoritma untuk mencapai *goal node*, dan banyaknya *url* unik yang di-*visit* oleh algoritma.

Dalam backend ini pula terdapat algoritma-algoritma yang kita pakai untuk melakukan pencarian, seperti BFS, IDS, Multi-BFS, dan Multi-DFS. Semua file yang dibuat dengan tujuan modularitas, digabung menjadi satu package, yaitu package *main*. Cara API menentukan penggunaan algoritma apa dalam pemrosesan *request* yang diterima adalah dengan membaca informasi *AlgoChoice* yang menjadi salah satu atribut *request* yang dikirimkan oleh frontend. *AlgoChoice* terdapat 4 macam sesuai dengan jenis algoritma apa yang ingin digunakan untuk melakukan pencarian.

3.4. Contoh Ilustrasi Kasus

Contoh kasus yang dapat diambil adalah misalkan pengguna ingin mencari jarak terdekat dari titik artikel *wikipedia* Joko Widodo (dengan *url* https://en.wikipedia.org/wiki/Joko_Widodo) menuju ke titik artikel *wikipedia* lain, misalnya Korea (dengan *url* https://id.wikipedia.org/wiki/Korea). Maka, pengguna *website* dapat memilih jenis aksi yang hendak dilakukan terhadap kedua informasi tersebut, yaitu dapat memilih antara menjalankan pencarian satu rute dengan IDS atau BFS, ataupun mencari banyak rute yang mungkin dengan MultiIDS atau MultiBFS.

Dengan algoritma BFS, proses pencarian dimulai dengan menempatkan URL artikel Joko Widodo sebagai *node* awal dalam antrian pencarian, yang dikelola menggunakan struktur data *'queue'*. Setiap *node* dalam *queue* ini disimpan sebagai *array of array of string*, yang juga mencakup jalur yang telah dilalui untuk mencapai node tersebut. Untuk memastikan setiap artikel hanya diproses sekali, digunakan *map* 'visited' dimana kunci adalah URL artikel, dan nilainya adalah *boolean* yang menandakan apakah artikel tersebut telah dikunjungi. Artikel Joko Widodo ditandai sebagai *'visited'* di awal proses untuk menghindari pengulangan dalam pencarian. Selanjutnya, dialkukan scraping terhadap halaman artikel Joko Widodo untuk membangkitkan semua *link* atau *node* anak yang terkait dengan artikel Joko Widodo. Ini melibatkan pengambilan semua URL yang terhubung dari halaman tersebut, yang mewakili kedalaman pertama atau level pertama dalam pencarian BFS. Tiap link atau node anak ini kemudian dievaluasi untuk menentukan apakah salah satunya adalah artikel target, yaitu Korea. Jika link yang dievaluasi tidak cocok dengan URL target dan belum pernah dikunjungi (berdasarkan *map 'visited*'), link tersebut akan ditambahkan ke *queue* baru, yang akan dijelajahi di kedalaman berikutnya. Proses ini berlanjut secara iteratif: mengambil *node* dari *queue*, melakukan *scraping* untuk mendapatkan *link* terkait, dan mengevaluasi tiap *link* tersebut. Untuk setiap *node* yang diproses, iterasi meliputi evaluasi kecocokan dengan target serta pengecekan status kunjungan untuk menghindari pengulangan. Dalam kasus BFS single path, algoritma akan segera menghentikan pencarian dan mengembalikan jalur yang ditemukan segera setelah menemukan *link* yang cocok dengan artikel Korea. Ini memastikan efisiensi dalam mencari jalur terpendek, karena pencarian berhenti ketika solusi pertama ditemukan. Sementara itu, dalam implementasi BFS multi path, ketika sebuah node cocok dengan target, jalur ke *node* tersebut disimpan tetapi pencarian terus berlanjut pada *level* kedalaman yang sama untuk menemukan kemungkinan jalur lain. Ini memungkinkan pengumpulan semua jalur potensial yang mencapai target pada kedalaman yang sama. Setelah semua *node* di *level* kedalaman tersebut diproses, semua jalur yang ditemukan kemudian dikembalikan sebagai hasil.

Dengan algoritma IDS, pertama artikel Joko Widodo akan disimpan pertama pada *array of string* `path` (atau kita sebut sebagai *stack* pada penjelasan ini). Karena perlakuan IDS, maka dilakukan iterasi pada *depth 0*, dan DLS pertama akan selesai dengan bukan solusi karena *starting node* bukanlah *goal node*. Namun, karena Joko Widodo telah dikunjungi, maka akan ditandai pada variabel *visited* yang menandakan bahwa *url* tersebut sudah pernah dikunjungi dan

ditandai dengan *boolean true*. Kemudian, sebelum dilanjutkan DLS iterasi 1 atau DLS dengan kedalaman 1, maka variabel *visited* akan di-*reset* kembali menjadi kosong karena DFS pada DLS nantinya akan dimulai lagi dari *starting node*. Pada saat perlakuan DLS dengan kedalaman 1, berbeda dengan *depth 0*, kali ini akan dilakukan *scraping url*, karena diperlukan *node-node* anak hasil pembangkitan dari *parent node / starting node*. Kemudian, link hasil *scraping* tersebut akan di-*cache* dan disimpan dengan *key url node* yang di-*scrap*. Lalu, dilakukan iterasi terhadap seluruh *url* dan dilakukan pengecekan apakah terdapat *node* yang merupakan *goal node* sehingga algoritma IDS dapat dihentikan dan melakukan pengembalian rute yang ditempuh oleh algoritma dari *starting node* untuk menuju *goal node*. Namun, jika tidak ditemukan *goal node*, maka pada DLS tingkat 2 nantinya akan dilakukan pembangkitan anak dari *link* yang diiterasi dan kemudian di-*scraping* lagi sehingga akar pencarian akan menjadi lebih lebar dan dalam. Hal tersebut dilakukan berulang-ulang hingga benar ditemukan solusi untuk mencapai *goal node*. Sebagai tambahan, jika ingin mencari seluruh kemungkinan rute dengan jarak terpendek, maka ketika algoritma pertama kali menemukan solusi atau *goal node*, iterasi tetap dilanjutkan hingga *node* saudaranya habis diiterasi dan dilakukan pengecekan hanya pada kedalaman yang sama. Dengan itu, maka akan didapatkan rute-rute yang mungkin untuk *depth* yang sama, yang berarti jarak atau banyak *node* yang ditempuh sama pula. Sebagai tambahan informasi, perlakuan *scraping* pada sebuah *node* terlebih dahulu dilakukan pengecekan *cache*. Jika *url* tersebut sudah pernah di-*scrap*, maka cukup mengambilnya dari *cache* dibandingkan melakukan *scraping* ulang. Terakhir, setiap *url* yang pertama kali di-*visit* akan disimpan ke dalam atribut *visited* sehingga algoritma tidak akan melakukan pada suatu *url* lebih dari satu kali dan akan mengoptimalkan waktu eksekusi program.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1. Spesifikasi Teknis Program

4.1.1. Struktur Data Program

a. Algoritma IDS

Pada algoritma IDS, struktur data yang digunakan sebagai parameter untuk melakukan penyimpanan *node-node* yang ditempuh untuk mencapai *goal node* disimpan hanya dalam *array of string* atau bisa dikatakan merupakan list biasa. Namun, karena algoritma IDS menggunakan mekanisme rekursi dalam pencariannya untuk mencari *goal node*, maka representasi pengeluaran dan pemasukan elemen ke dalam list, sama seperti sifat struktur data *stack*.

Algoritma IDS ini melakukan DLS secara berulang dengan penambahan *depth* secara progresif hingga menemukan solusi atau *path* untuk mencapai *goal node*. Jadi, dalam setiap iterasinya, algoritma berjalan dengan DLS kedalaman 1, 2, 3, hingga n, yang merupakan kedalaman dimana algoritma pertama kali mencapai *goal node*. Dalam setiap DLS juga benar-benar dilakukan pencarian dari *starting node*, atau dengan kata lain melakukan DFS secara progresif pula.

Dalam proses DFS ini pula terlihat adanya pemasukan dan pengeluaran elemen dari *array of string `path`* yang menyimpan informasi *node-node* yang ditempuh oleh algoritma untuk mendapatkan solusi, dimulai dari *starting node* hingga *goal node*. Jadi, terdapat dua jenis kemungkinan, (1) jika pada sebuah DLS belum menemukan solusi dan masih belum mencapai kedalaman maksimal DLS tersebut, maka algoritma akan menambahkan *node* tersebut ke dalam *path* (perlakuan *push* secara implisit) sehingga akan dilakukan pencarian lagi ke *depth* lagi yang dalam hingga mencapai batas *depth* DLS tersebut dan (2) jika pada sebuah DLS belum menemukan solusi namun sudah mencapai kedalaman maksimal DLS tersebut dan belum semua *node* dilakukan iterasi, atau masih ada kemungkinan *node* yang lain dapat dikunjungi, maka algoritma akan melakukan *backtracking* atau mundur sehingga *path* akan melakukan *pop* secara tidak langsung (tanpa melakukan pemanggilan secara eksplisit).

Oleh karena itu, berdasarkan penjelasan pada paragraf sebelumnya, dapat diketahui bahwa sebenarnya IDS merupakan iterasi DLS secara progresif hingga menemukan solusi yang tepat, yang berarti penerapan struktur data pada algoritma DLS juga merupakan struktur data yang sebenarnya diterapkan oleh IDS. Sebagai penekanan, struktur data yang digunakan merupakan *array of string* dalam algoritma IDS ini memiliki sifat mirip dengan *stack*.

b. Algoritma BFS

Pada algoritma BFS, queue adalah komponen kunci yang digunakan untuk memfasilitasi pencarian secara terstruktur dan sistematis melalui graf . Di dalam konteks BFS, queue memungkinkan pencarian berlangsung secara level demi level, dimulai dari node awal dan bergerak maju ke node-node yang berdekatan hingga menemukan target atau telah mengeksplorasi semua kemungkinan jalur. Proses ini dimulai dengan menambahkan URL awal ke dalam queue, dianggap sebagai node awal dari pencarian dan mewakili tingkat kedalaman pertama. Queue ini memainkan peran penting dalam memastikan bahwa pencarian dilakukan secara breadth-first, artinya algoritma menjelajahi semua node pada satu kedalaman sebelum melanjutkan ke kedalaman berikutnya.

Kemudian digunakan struktur data map untuk merekam node-node yang telah dikunjungi. Kunci dalam map adalah URL, dan nilai adalah boolean yang menandakan bahwa URL tersebut telah dikunjungi. Struktur data ini mencegah algoritma dari memproses node yang sama lebih dari satu kali, yang sangat penting untuk menghindari siklus dan meminimalisir overhead komputasi.

Setiap iterasi algoritma melibatkan mengambil node dari queue untuk dieksplorasi. Proses eksplorasi ini mencakup scraping URL saat ini untuk menemukan semua URL terkait yang kemudian diperiksa keberadaannya dalam map visited. URL yang belum dikunjungi ditandai sebagai dikunjungi di dalam map visited dan ditambahkan ke dalam queue, siap untuk dijelajahi pada kedalaman berikutnya. Ini meminimalkan redundansi dan pengulangan yang tidak perlu, meningkatkan efisiensi algoritma dengan menghindari siklus.

Ketika semua node pada tingkat kedalaman tertentu telah diproses, queue yang diperbarui akan berisi node untuk tingkat kedalaman berikutnya, mengatur transisi algoritma dari satu lapis ke lapis berikutnya dalam graf. Proses ini terus berlanjut sampai URL target ditemukan atau semua kemungkinan jalur telah tuntas dieksplorasi. Dalam hal pencarian multi-path, pencarian berlanjut di tingkat kedalaman yang sama hingga semua jalur yang mungkin telah diidentifikasi, sebelum berhenti atau melanjutkan lebih dalam.

Selain itu, struktur data dalam algoritma ini memungkinkan setiap node dalam queue untuk menyimpan jalur lengkap dari node awal hingga node saat ini. Hal ini tidak hanya memungkinkan penyimpanan langsung dari jalur yang berhasil mencapai target tetapi juga memudahkan rekonstruksi jalur lengkap untuk setiap node yang diproses. Ini memungkinkan algoritma untuk dengan mudah merekonstruksi jalur lengkap kembali ke node asal saat target ditemukan.

Kesimpulannya, pengelolaan queue yang cermat, penggunaan map untuk melacak kunjungan, serta penyimpanan efisien dari jalur yang dilalui, semuanya berkontribusi pada efektivitas BFS dalam menemukan jalur terpendek atau semua jalur potensial dalam graf yang luas dan kompleks.

4.1.2. Fungsi dan Prosedur Program

Dalam keberjalan program DFS dan BFS yang telah dijelaskan sebelumnya, maka dibutuhkan fungsi dan prosedur untuk memenuhi kebutuhan tersebut. Maka dari itu, terdapat implementasi kedua algoritma tersebut dalam bentuk fungsi yang akan dilampirkan dalam bentuk *pseudocode* sebagai berikut.

a. Fungsi DLS

```
function DLS(depth: integer, startLink, goalLink: string,
currentDepth: integer, path: array of string, visited: map
of string -> bool, cache *LocalCache) {
    if currentDepth > depth {
        -> error
    }

    Tambahkan visited[startLink] = true untuk menandai
bahwa startLink sudah dikunjungi

    if (startLink berada di dalam cache) {
```

```
        ambil hasil scraping dari cache
    } else {
        ambil hasil scraping dari metode fetchLinks dengan
menggunakan library gocolly
        simpan hasil scraping ke dalam cache
    }

    Data scraping disimpan pada variabel links

    for link in (links) {
        if (link tidak pernah divisit) {
            visited[link] = true
        } else {
            continue
        }

        if (link == goalLink) {
            -> append(path, link)
        } else {
            new_path = append(path, link)
            result = DLS(depth, link, goalLink,
currentDepth+1, new_path, visited, cache)
            if (result tidak bukan error) {
                -> result
            }
        }
    }

    Kembalikan error tidak menemukan hasil
}
```

b. Fungsi IDS

```
function IDS(startLink, goalLink: string){

    Inisialisasi cache

    while (true) {
        Inisialisasi map visited

        result = DLS(i, startLink, goalLink, 0,
[]string{startLink}, visited, cache)

        if (tidak ada error) {
            -> solusi
        } else {
            tidak ditemukan solusi
        }
```

```
        Batasi iterasi sebanyak 20 kali
    }

    -> Error tidak menemukan goal
}
```

c. Fungsi Multi-DLS

```
function MultiDLS(depth: integer, startLink, goalLink:
string, currentDepth: integer, path: array of string,
visited: map of string -> bool, cache *LocalCache,
multiple_path_save: pointer to array of array of string)
{
    if currentDepth > depth {
        -> error
    }

    Tambahkan visited[startLink] = true untuk menandai
bahwa startLink sudah dikunjungi

    if (startLink berada di dalam cache) {
        ambil hasil scraping dari cache
    } else {
        ambil hasil scraping dari metode fetchLinks
dengan menggunakan library gocolly
        simpan hasil scraping ke dalam cache
    }

    Data scraping disimpan pada variabel links

    for link in (links) {
        if (link tidak pernah divisit) {
            visited[link] = true
        } else {
            continue
        }

        if (link == goalLink) {
            Masukkan / append link ke multiple_path_save
        } else {
            new_path = append(path, link)

            {* Panggil Rekursi *}
            MultiDLS(depth, link, goalLink,
currentDepth+1, new_path, visited, cache,
multiple_path_save)
        }
    }
```

```
    if (multiple_path_save sudah memiliki solusi) {
        hentikan rekursi
    }

    -> error tidak menemukan goal node
}
```

d. Fungsi Multi-IDS

```
function MultiIDS(startLink, goalLink: string){

    Inisialisasi cache
    Inisialisasi multiple_path_save

    while (true) {
        Inisialisasi map visited

        result = DLS(i, startLink, goalLink, 0,
[]string{startLink}, visited, cache, &multiple_path_save)

        if (tidak ada error) {
            -> multiple_path_save
        } else {
            tidak ditemukan solusi
        }

        Batasi iterasi sebanyak 20 kali
    }

    -> Error tidak menemukan goal
}
```

e. Fungsi BFS

```
function bfs(startURL, targetURL: string){
    Inisialisasi startTime
    Inisialisasi sync : wg, mu
    Inisialisasi map visited
    visited[startURL] <- True
    numChekced <- 0
    Inisialisasi array of array string bernama queue
    Inisialisasi channels : results, done
    maxGoroutines <- 20

    DEFINE processURls(paths)
        Inisialisasi array of array strnig bernama
localQueue
        FOR each path in paths
```

```
            currentURL <- path[ len(path) -1 ]
            articleRequested += 1
            links, err <- scrapeWikipediaLink(currentURL)

            if (err ) {
                output ("Error Scrapping :",err)

            FOR each link in links
                  numchecked += 1
                  if (link == targetURL) {
                       result.append(path,link)
                       return

                  if (! visited[link]){
                        visited[link] <- True
                        newPath.append(path…)
                        newPath.append(link)
                        localQueue.append(newPath)
            queue.append(localQueue…)

     START Goroutine
         FOR len(queue) > 0
             currentBatch <- queue
             queue <- nil

             for each paths in currentBatch
                  acquire semaphore
                  add task to WaitGroup
                  mulai goroutine dengan processURLS
untuk path group
              Tunggu semua goroutines selesai
      TUTUP channel
      END Goroutine

      IF path received from results
           duration <- startTime- currentTime
           return path, pathlength, numchecked,
articleRequested, duration
      IF done channel closed
           return error "no path found"
```

f. Fungsi BFS Multi path

```
function bfsMultiPath(startURL, targetURL: string){
    Inisialisasi startTime
    Inisialisasi sync : wg, mu
    Inisialisasi map visited
    visited[startURL] <- True
    numChekced <- 0
```

```
    Inisialisasi array of array string bernama queue
    Inisialisasi map array of array string bernama
resultsMap
    Inisialisasi channels : results, done
    foundLevel <- -1
    maxGoroutines <- 20
    goroutineSem <- Inisialisasi channel  dengan ukuran
maxGoroutines

    FOR len(queue) > 0
        currentLevel <- queue
        queue <- nil
        Inisialisasi array of array string bernama
levelQueue

        FOR each paths in currentBatch
            add task to WaitGroup
            mulai goroutine

            START Goroutine
                currentURL <- path[ len(path) -1 ]
                IF foundlevel ≠ -1 && len(path) >
foundLevel
                      return

                links, err <-
scrapeWikipediaLink(currentURL)

            if (err ) {
                output (“Error Scrapping :”,err)

            FOR each link in links
                  numchecked += 1
                  if (link == targetURL && (foundLevel ==
-1 OR len(path) ==  foundLevel)) {
                     pathSignature <-
createPathSignature(append(path,link))
                     if signature not in resultsMap
                         resultsMap[pathSignature] <-
newPath
                         results.append(newPath)
                      foundlevel <- len(path)
                  if (foundLevel ≠ -1 && len(path) ≥
foundLevel){
                       continue
                  if (! visited[link]){
                       visited[link] <- True
                        levelQueue.append(path,link) …)
            queue  <- levelQueue

     duration <- startTime- currentTime
```

```
     IF (len(results)>0) {
          pathlength<- len(results[0])
          return results, pathlength, numchecked,
duration, nil
     return nil, 0 , numChecked, duration, err "no path
found"
```

4.2. Tata Cara Penggunaan Program

Pastikan *Docker* telah terinstal di sistem Anda. Jika belum, lakukan instalasi *Docker* sesuai dengan petunjuk resmi dari situs *web Docker*. Jika *Docker* sudah terinstal lakukan hal berikut:

1. Silahkan lakukan *clone repository* ini dengan cara menjalankan perintah berikut pada terminal

   ```
   git clone https://github.com/ChaiGans/Tubes2_BarengApin.git
   ```

2. Jalankan perintah berikut pada terminal untuk memasuki root directory program

   ```
   cd ./Tubes2_BarengApin
   ```

3. Pastikan Docker Desktop telah terbuka. Setelah pengguna berada pada root directory, jalankan perintah berikut pada terminal

   ```
   docker compose build
   ```

   Jika docker compose build telah selesai, jalankan perintah berikut

   ```
   docker compose up
   ```

4. Untuk menjalankan website masuk ke link berikut ini

   ```
   http://localhost:3000
   ```

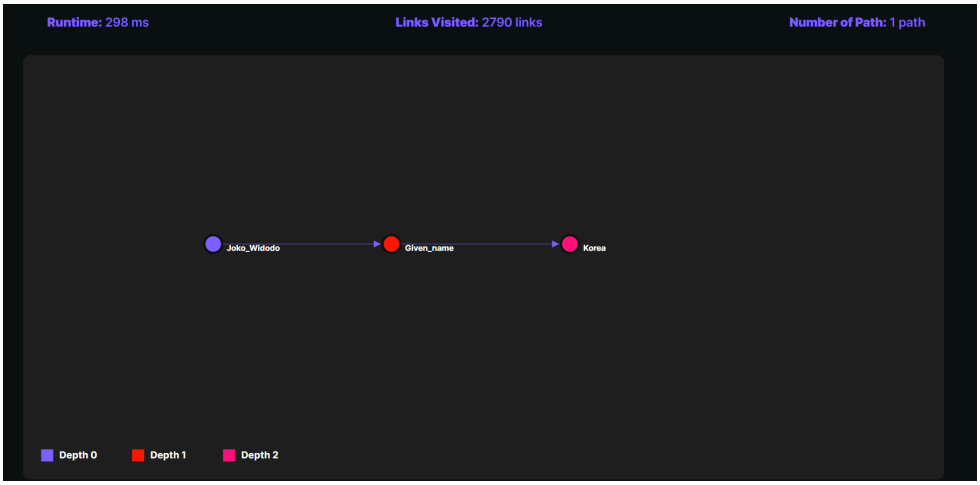
5. Setelah pengguna berhasil menjalankan website, pengguna dapat memilih algoritma pencarian, baik dengan menggunakan algoritma BFS maupun IDS.

6. Kemudian, pengguna dapat memilih untuk mencari pengeluaran *single-path* atau *multi-path*.
7. Setelah pengguna memilih algoritma pencarian dan tipe keluaran, pengguna memilih judul artikel Wikipedia asal dan tujuan. Program juga akan memberikan rekomendasi artikel Wikipedia berdasarkan judul yang dimasukkan oleh pengguna agar pengguna memiliki pilihan untuk memilih rekomendasi tersebut.
8. Program akan menampilkan rute terpendek antara kedua artikel dalam bentuk visualisasi graf, beserta waktu eksekusi, jumlah artikel yang dilalui, kedalaman pencarian, serta detail jalur seperti tampilan gambar untuk masing-masing tautan dan dikelompokkan berdasarkan tiap jalur.

4.3. Hasil Pengujian

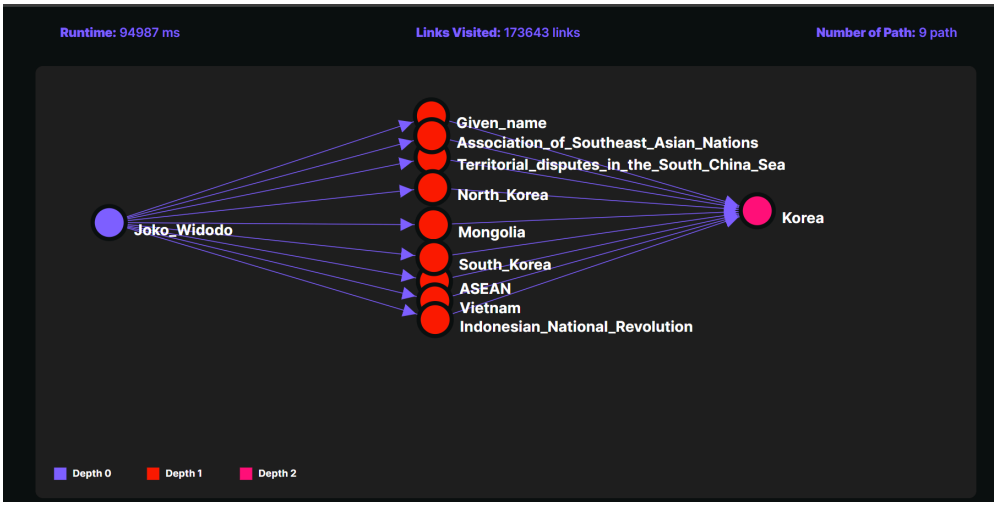
4.3.1. *Test Case* 1 (Derajat 2)

a. *BFS Single path*



Gambar 4.3.1.1 Test Case BFS Single Path I (Derajat 2)

b. *BFS Multi path*

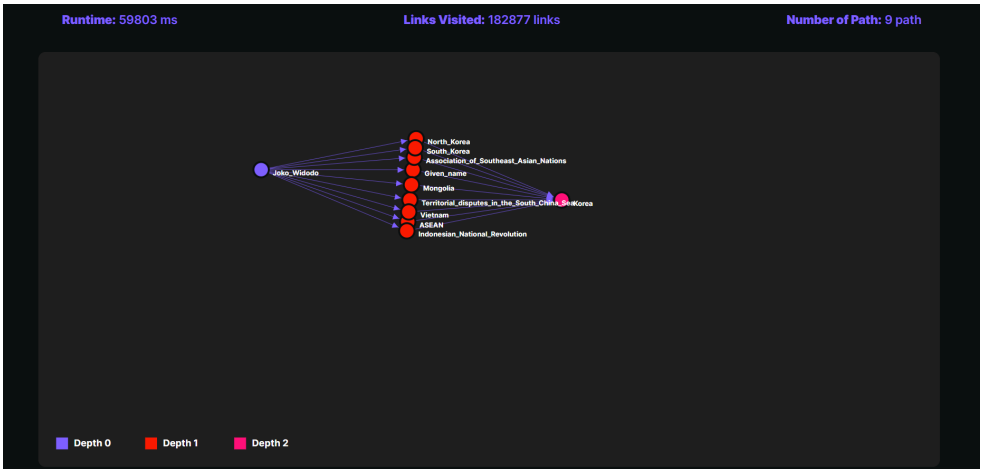


Gambar 4.3.1.2 Test Case BFS Multi Path I (Derajat 2)

c. ***IDS Single path***

Gambar 4.3.1.3 Test Case IDS Single Path I (Derajat 2)

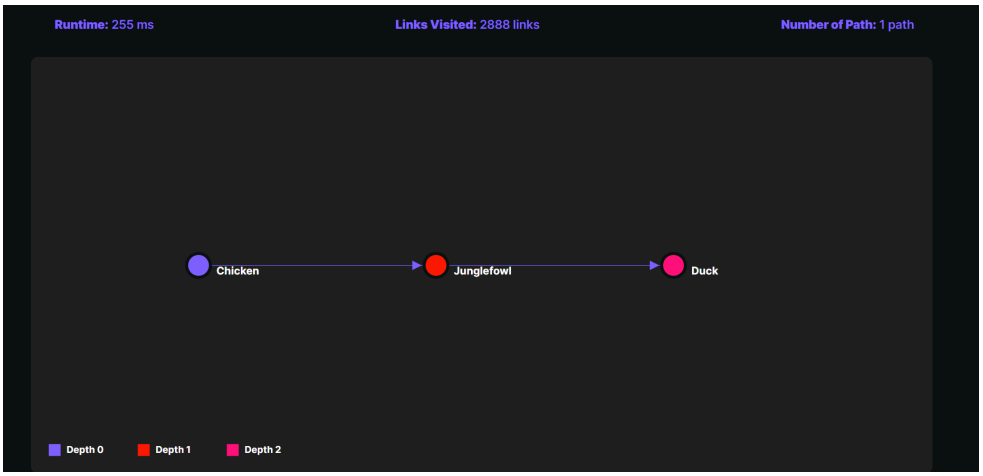
d. ***IDS Multi path***



Gambar 4.3.1.4 Test Case IDS Multi Path I (Derajat 2)

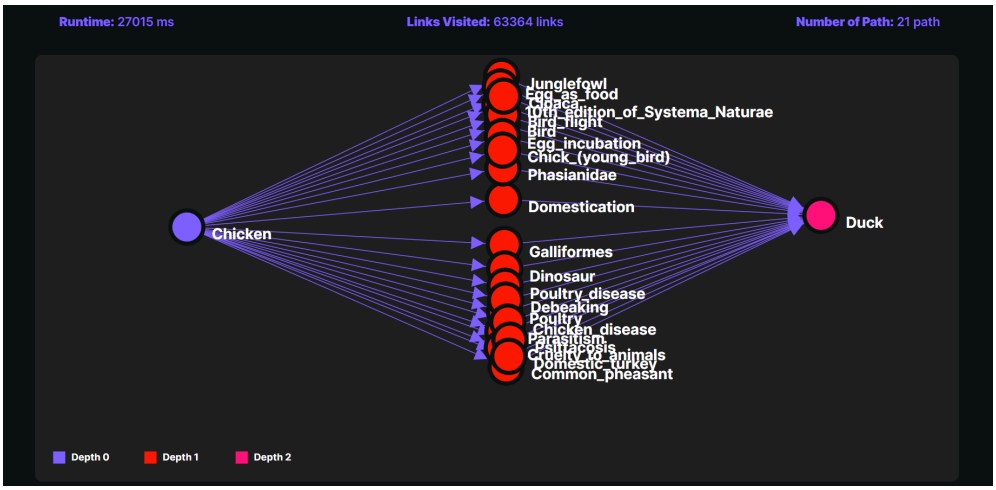
4.3.2. Test Case 2 (Derajat 2)

a. BFS *Single path*



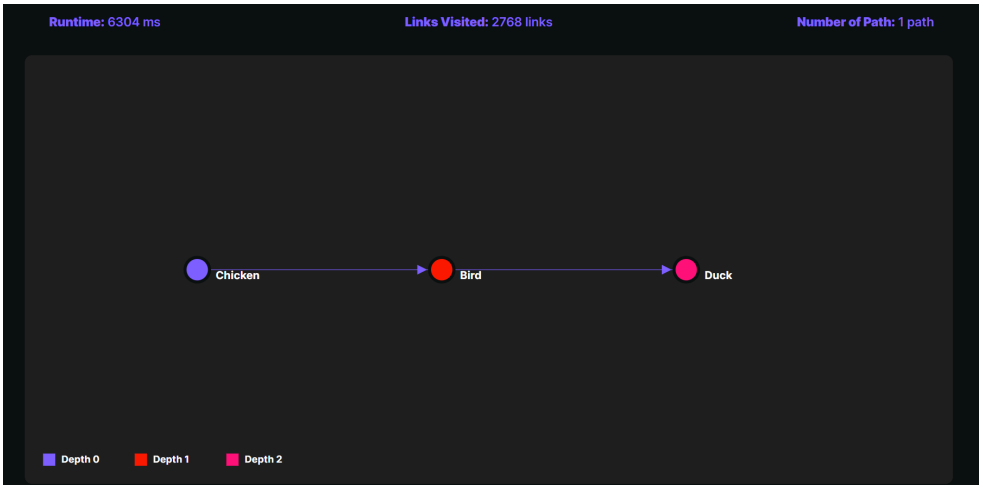
Gambar 4.3.2.1 Test Case BFS Single Path II (Derajat 2)

b. *BFS multi path*



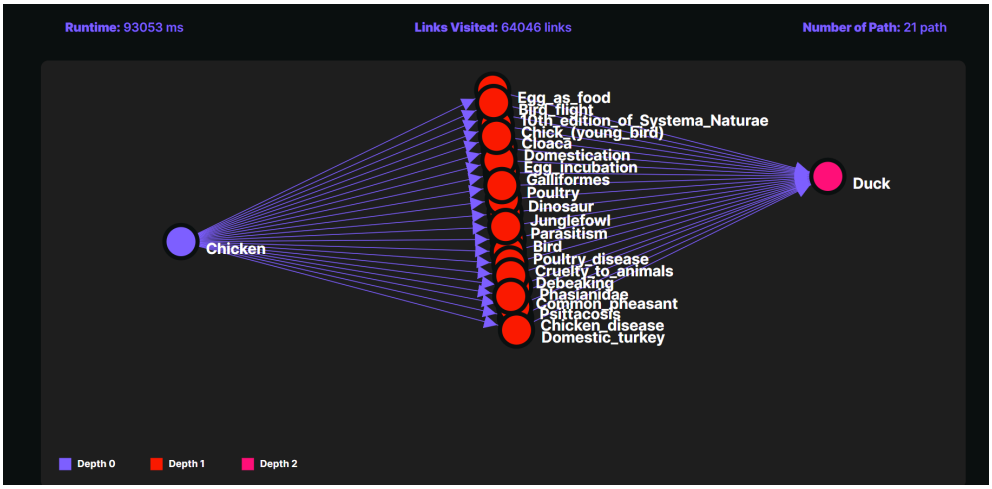
Gambar 4.3.2.1 Test Case BFS Multi Path II (Derajat 2)

c. ***IDS single path***



Gambar 4.3.2.3 Test Case IDS Single Path II (Derajat 2)

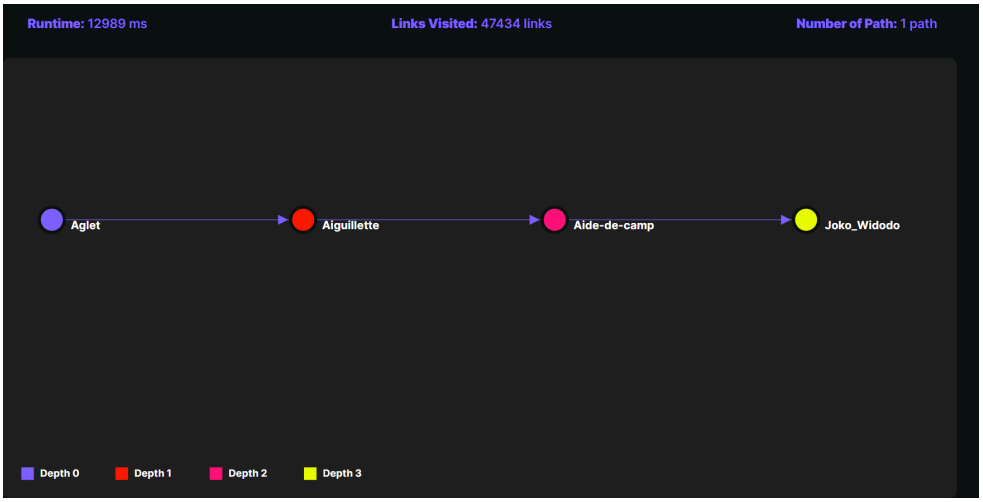
d. ***IDS multi path***



Gambar 4.3.2.4 Test Case IDS Multi Path II (Derajat 2)

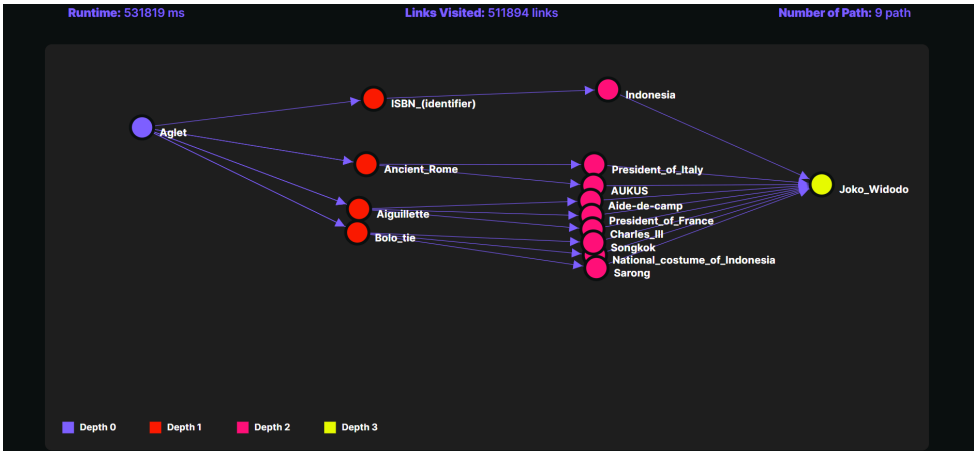
4.3.3. *Test Case* 3 (Derajat 3)

a. ***BFS Single path***



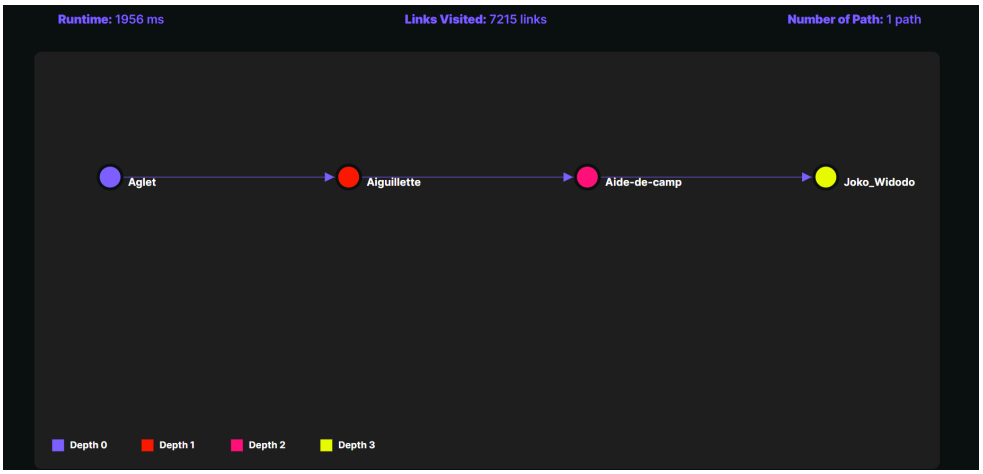
Gambar 4.3.3.1 Test Case BFS Single Path III (Derajat 3)

b. ***BFS multi path***



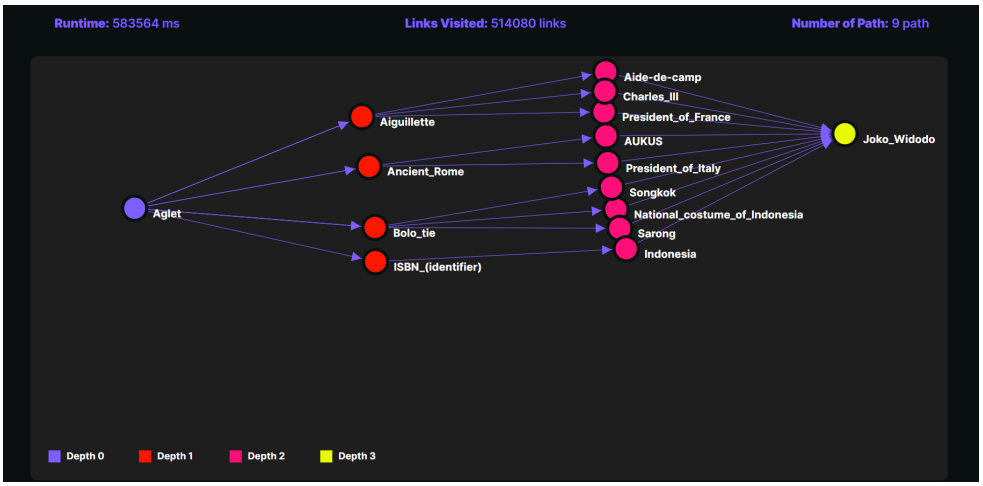
Gambar 4.3.3.2 Test Case IDS Single Path III (Derajat 3)

c. ***IDS single path***



Gambar 4.3.3.3 Test Case IDS Single Path III (Derajat 3)

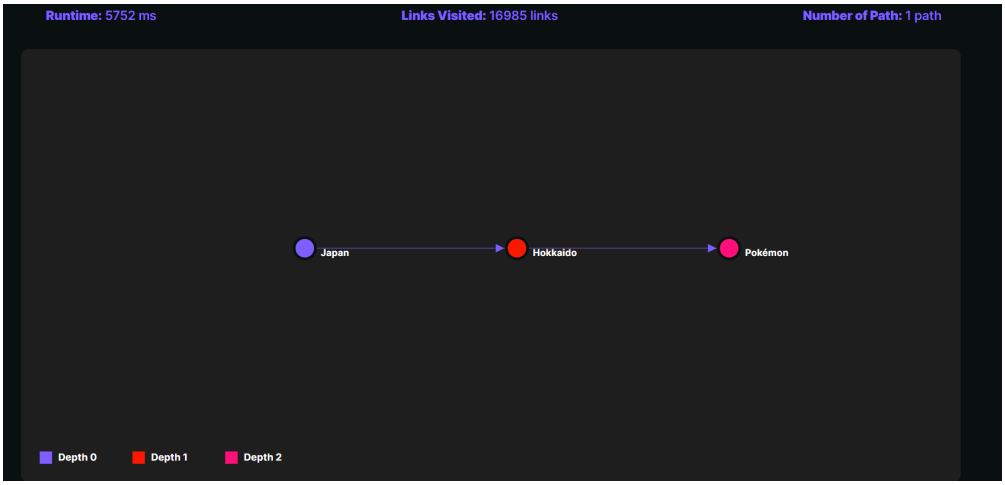
d. ***IDS multi path***



Gambar 4.3.3.4 Test Case IDS Multi Path III (Derajat 3)

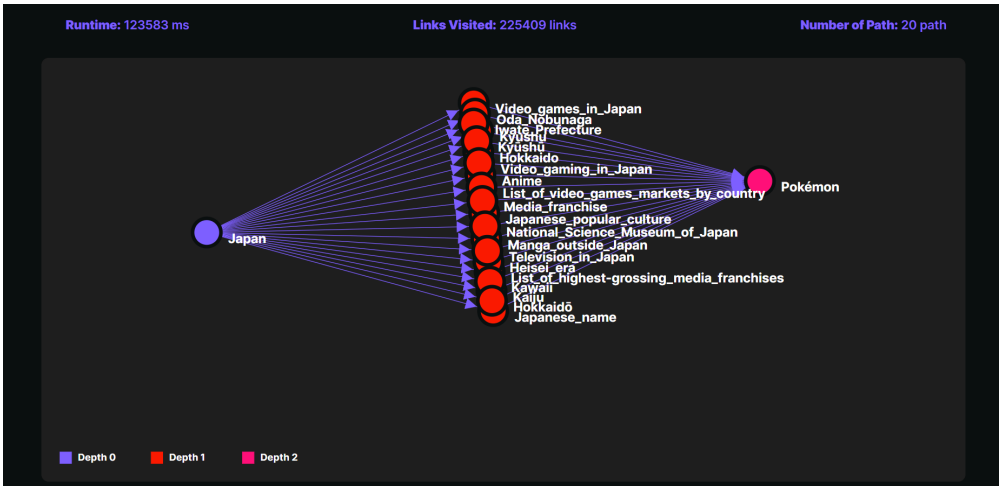
4.3.4. Test Case 4 (Derajat 2)

a. ***BFS single path***



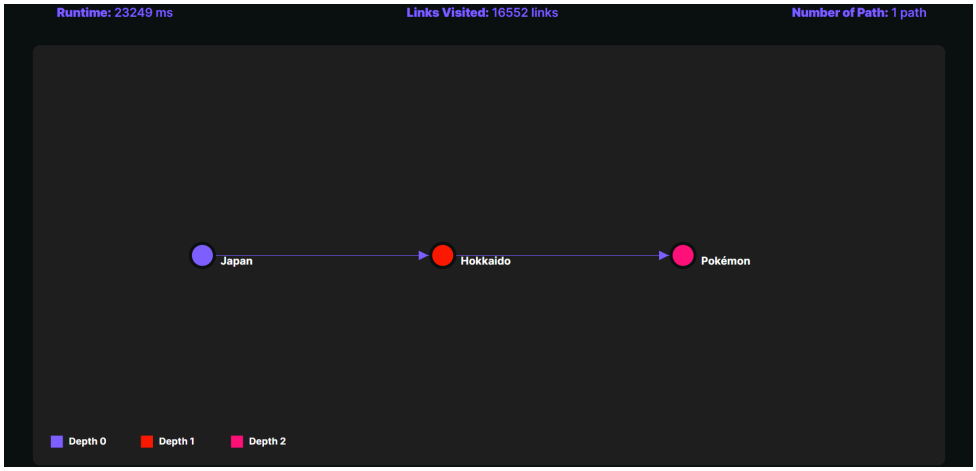
Gambar 4.3.4.1 Test Case BFS Single Path IV (Derajat 2)

b. ***BFS multi path***



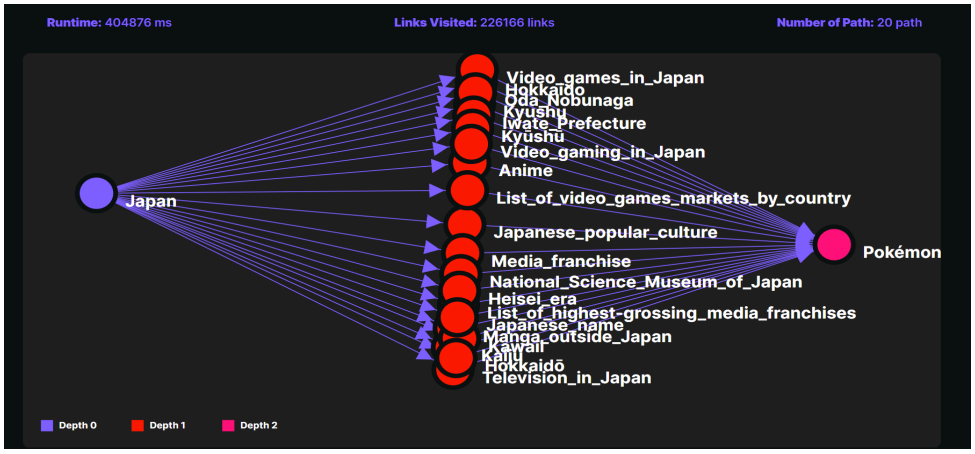
Gambar 4.3.4.2 Test Case BFS Multi Path IV (Derajat 2)

c. ***IDS single path***



Gambar 4.3.4.3 Test Case IDS Single Path IV (Derajat 2)

d. ***IDS multi path***



Gambar 4.3.4.4 Test Case IDS Multi Path IV (Derajat 2)

4.4. Analisis Hasil Pengujian

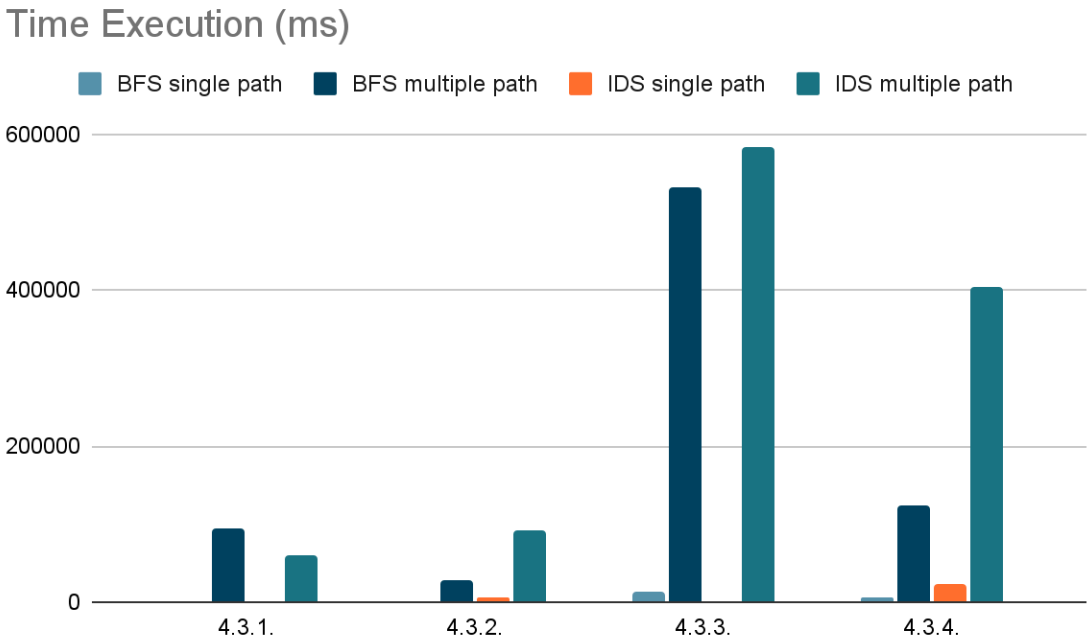
Dari hasil pengujian yang telah dilakukan pada **bagian 4.3**. dengan melakukan eksperimen langsung pada program dengan kasus yang berbeda-beda, didapatkan data-data yang dapat divisualisasikan dalam tabel sebagai berikut.

Tabel 4.4.1. Visualisasi data hasil pengujian dalam bentuk tabular

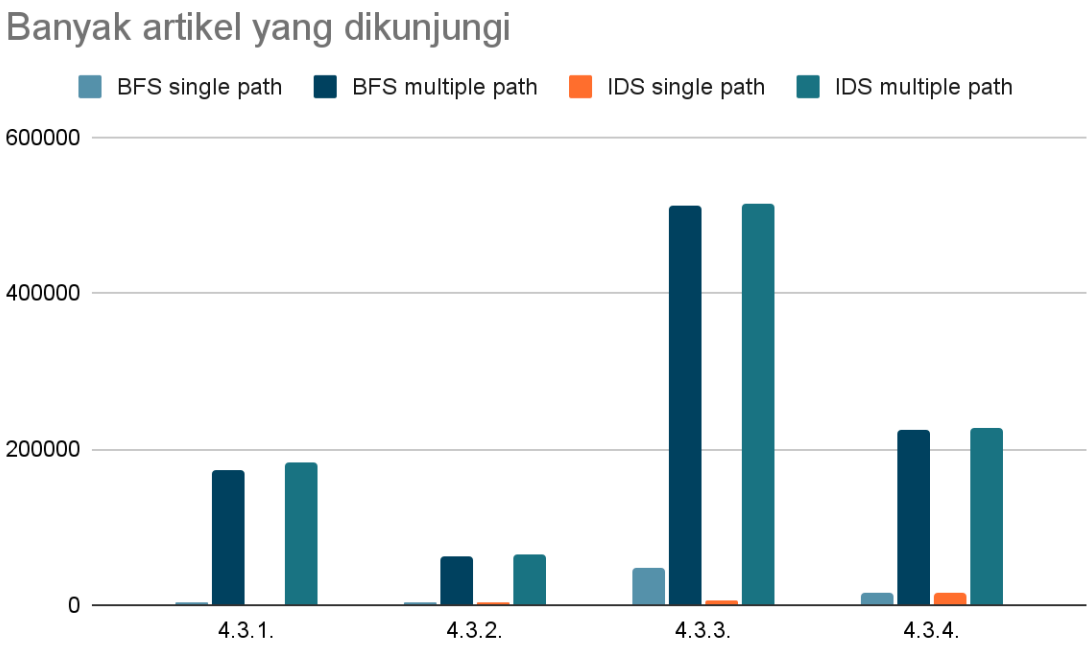
Nomor *Test Case*	Jenis algoritma	Banyak artikel dikunjungi (*url*)	Waktu eksekusi (ms)	Banyak kedalaman (derajat)
4.3.1.	*BFS single path*	2790	298	2
	BFS multiple path	173643	94987	
	IDS single path	1185	481	
	IDS multiple path	182877	59803	
4.3.2.	*BFS single path*	2888	255	2
	BFS multiple path	63364	27015	
	IDS single path	2768	6304	
	IDS multiple path	64046	93053	
4.3.3.	*BFS single path*	47434	12989	3
	BFS multiple path	511894	531819	
	IDS single path	7215	1956	
	IDS multiple path	514080	583564	
4.3.4.	*BFS single*	16985	5752	2

	path			
	BFS multiple path	225409	123583	
	IDS single path	16552	23249	
	IDS multiple path	226166	404876	

Dari hasil data yang dapat dibaca secara tabular pada **Tabel 4.4.1.** dapat divisualisasikan menjadi grafik sehingga dapat didapatkan korelasi yang lebih jelas serta keterhubungan dan atribut yang konsisten dalam setiap algoritma. Terdapat dua atribut yang dapat dianalisis, yaitu berupa jumlah artikel yang dilalui oleh algoritma untuk mencapai solusi serta waktu eksekusi program yang dibutuhkan untuk mencapai solusi yang diinginkan. Kedua visualisasi tersebut dapat dilihat pada **Gambar 4.4.1.** dan **Gambar 4.4.2.**



Gambar 4.4.1. Visualisasi waktu eksekusi dari data tabular



Gambar 4.4.2. Visualisasi banyak artikel yang dikunjungi dari data tabular

Berdasarkan visualisasi informasi data waktu eksekusi algoritma, terdapat beberapa informasi yang dapat diperoleh terkait korelasi dan perbandingan antara satu algoritma dengan yang lainnya. Agar perbandingan atau komparasi yang dilakukan bersifat setara atau *apple to apple*, maka perbandingan terlebih dahulu dilakukan dengan melakukan perbandingan algoritma pencarian solusi tunggal, yaitu algoritma IDS *Single-Path* dan BFS *Single-Path*. Terlihat bahwa dari empat kali pengujian yang dilakukan bahwa BFS *single-path* mengungguli waktu eksekusi dalam pencarian solusi pertama sebanyak tiga kali, tepatnya pada pengujian ***test case 4.3.1, 4.3.2, serta 4.3.4.*** dan kebetulan sekali bahwa ketiga *test case* tersebut merupakan pengujian dengan pencarian berderajat dua. IDS mengungguli sebanyak satu kali pada ***test case 4.3.3.*** dengan perbedaan waktu eksekusi yang cukup signifikan.

Secara heuristik, atau dengan logika pemikiran, seharusnya BFS akan selalu mengungguli IDS karena IDS terlalu banyak membuang waktunya untuk melakukan iterasi secara berulang karena IDS merupakan algoritma yang melakukan iterasi IDS dengan *depth* yang bertambah secara progresif hingga solusi pertama ditemukan. Namun, di sisi lain, terdapat beberapa kasus dimana IDS masih dapat mengungguli BFS seperti yang terdapat ***test case 4.3.3***., yang memiliki *degree* yang lebih tinggi dibandingkan dengan *test case* lainnya. IDS memang merupakan sebuah algoritma yang lebih unggul dibandingkan dengan BFS jika berurusan dengan persoalan yang memiliki solusi dengan kedalaman yang cukup “dalam”, karena IDS cenderung untuk mencari solusi secara vertikal sehingga pencarian solusi yang *goal node* yang berada pada *depth* yang dalam juga akan diungguli oleh IDS.

Selain itu, untuk pencarian dengan *goal node* yang berada pada posisi yang jauh dari *starting node* secara kedalaman, akan menyebabkan BFS membutuhkan memori ruang yang sangat besar karena BFS akan cenderung bergerak *forward* dan menyimpan seluruh informasi, yang mana akan menyebabkan kompleksitas ruang yang sangat besar dibandingkan IDS yang tidak melakukan penyimpanan (karena melakukan mekanisme *backtracking* jika tidak menemukan solusi). Hal tersebut juga ditandai dengan banyak link yang dikunjungi oleh algoritma BFS ketika bertemu dengan *test case degree 3*, yang jumlahnya memiliki perbedaan yang cukup signifikan dengan algoritma IDS.

Selain komparasi untuk performansi kedua algoritma pencarian untuk solusi tunggal, terdapat fakta menarik bahwa waktu eksekusi dan banyak artikel yang dikunjungi

menunjukkan angka yang hampir setara dengan perbedaan yang sangat kecil dibandingkan algoritma pencarian solusi tunggal. Ini karena kedua algoritma tersebut sama-sama melakukan *traversal graf* secara keseluruhan tanpa pengecualian, dan tidak ada yang namanya faktor keberuntungan dalam pencarian akibat bertemunya solusi terlebih dahulu akibat cara perlakuan *traversal* algoritma. Dengan informasi tersebut pula, kita dapat membuktikan bahwa algoritma IDS dan BFS pada akhirnya memiliki tujuan yang sama, namun hanya dengan metode yang berbeda, dan juga membuktikan perlakuan *node* yang konsisten pada setiap persoalan. Hal tersebut memang dapat dilihat pada banyak artikel yang dikunjungi, namun berbeda dengan waktu eksekusi yang dilakukan oleh kedua algoritma tersebut.

Jika dipikir secara logika berdasarkan pendekatan yang dijelaskan sebelumnya terkait kompleksitas ruang, bukankah seharusnya *BFS multipath* akan memiliki waktu eksekusi yang lebih lambat dibandingkan IDS *multipath* akibat kedalaman yang cukup "dalam" menyebabkan kompleksitas ruang yang tinggi. Jawabannya seharusnya benar, *BFS multipath* seharusnya memang lebih lambat untuk *degree* yang tinggi, namun terdapat faktor bahwa karena pada program dalam tugas besar kali ini, digunakan pemrosesan secara paralel pada algoritma BFS dengan memanfaatkan algoritma *goroutines*, sehingga algoritma BFS memiliki pemrosesan yang lebih unggul dibandingkan dengan IDS yang hanya memanfaatkan optimasi dengan mekanisme *caching*. Namun, secara teoretis, jika BFS juga dilakukan tanpa asinkronisasi, *time execution* seharusnya akan lebih tinggi dibandingkan dengan IDS pada kedalaman *goal node* yang lebih tinggi.

Selain perlakuan komparasi, terdapat pula anomali yang dapat dilihat dari visualisasi data bahwa bertambahnya jumlah artikel yang dikunjungi tidak searah dengan bertambahnya waktu eksekusi, seperti yang terlihat pada ***test case 4.3.1.*** bahwa IDS yang mengunjungi artikel yang lebih sedikit dibandingkan BFS namun memiliki waktu eksekusi yang lebih lama. Hal ini bisa jadi disebabkan oleh faktor yang disebutkan sebelumnya, yaitu pengaruhnya penggunaan optimasi asinkronisasi pada BFS sehingga memiliki waktu eksekusi yang lebih cepat meskipun *node* yang dibangkitkan lebih banyak.

BAB V
KESIMPULAN, SARAN, DAN REFLEKSI

5.1 Kesimpulan

Sebuah aplikasi *website* telah dikembangkan sebagai bagian dari tugas besar IF2211 Strategi Algoritma. Aplikasi ini bertujuan untuk menemukan lintasan terpendek dari satu artikel Wikipedia ke artikel lainnya menggunakan algoritma pencarian BFS (*Breadth-First Search)* dan IDS (*Iterative Deepening Search*). Pengembangan aplikasi *website* ini melibatkan penggunaan framework *NextJS* untuk bagian frontend dan *TailwindCSS* untuk mempermudah pembangunan UI website, sementara backendnya menggunakan bahasa pemrograman *Go* dengan framework *GIN*. Semua detail arsitektur, fungsionalitas, dan spesifikasi program telah dijelaskan pada bagian sebelumnya.

Berdasarkan hasil analisis yang telah dilakukan pada BAB IV dalam isi laporan ini, terdapat beberapa kesimpulan yang dapat diperoleh. Dalam perbandingan kinerja atau performansi algoritma pencarian IDS dan BFS untuk solusi tunggal maupun solusi banyak, didapatkan informasi bahwa BFS memiliki keunggulan waktu eksekusi dalam pencarian solusi pertama pada kebanyakan pengujian, terutama pada test case dengan pencarian berderajat rendah. IDS mengungguli BFS dalam satu pengujian, terutama pada test case dengan degree yang lebih tinggi. Secara teori, BFS cenderung lebih lambat dari IDS untuk degree yang tinggi karena kompleksitas ruang yang tinggi. Namun, dalam implementasi ini, BFS memperoleh keunggulan waktu eksekusi karena penggunaan pemrosesan paralel dengan *goroutines*. Kedua algoritma BFS dan IDS juga memiliki banyak artikel yang dikunjungi dengan angka yang cukup mirip sehingga menandakan bahwa kedua algoritma benar-benar melakukan iterasi seluruh *graf node* yang mungkin untuk dilalui.

Analisis yang dilakukan masih dipengaruhi oleh beberapa faktor eksternal, seperti pengaruh optimasi yang menyebabkan beberapa informasi yang menunjukkan anomali yang dapat dilihat dalam visualisasi data yang telah dilampirkan. Penggunaan optimasi seperti asinkronisasi pada BFS mempengaruhi waktu eksekusi, membuatnya lebih cepat meskipun mengunjungi lebih banyak artikel. Anomali terjadi pada beberapa pengujian, di mana jumlah artikel yang dikunjungi tidak selalu sejalan dengan waktu eksekusi. Ini dapat disebabkan oleh faktor-faktor seperti penggunaan optimasi atau karakteristik khusus dari setiap *test case*.

Dengan demikian, dari hasil pengujian yang telah dilakukan, terdapat sebuah informasi yang setidaknya dapat divalidasi dan didukung dengan hasil eksperimen yang cukup konsisten, yaitu bahwa IDS akan cenderung memiliki waktu eksekusi yang lebih cepat dibandingkan dengan BFS dengan alasan kompleksitas ruang. Namun, jika hasil dari eksperimen atau tugas besar ini untuk menentukan BFS atau IDS, mana yang lebih baik, kelompok BarengApin tidak dapat menentukan secara mutlak, dan lebih memilih hanya untuk menyarankan untuk menggunakan algoritma BFS untuk pencarian yang secara heuristik terdapat kemungkinan bahwa *goal node* berada di dekat dengan *starting node* dan disarankan untuk menggunakan algoritma IDS jika secara heuristik diketahui bahwa *goal node* akan berada jauh dari *starting node* atau dengan kata lain *goal node* berada di dalam kedalaman yang cukup "dalam" sehingga dapat menghemat kompleksitas ruang.

5.2 Saran

Untuk meningkatkan kinerja secara signifikan, disarankan untuk mengadopsi koneksi internet yang lebih stabil dan cepat. Hal ini akan memastikan bahwa proses *scraping* dapat berjalan lebih lancar dan jarang mengalami kesalahan

Selain itu, pemanfaatan *database* untuk menyimpan semua data artikel Wikipedia dapat menjadi solusi yang sangat efektif. Dengan menyimpan data secara terpusat, setiap kali pencarian dilakukan, informasi dapat diambil langsung dari database. Hal ini tidak hanya menghemat waktu dan penggunaan internet, tetapi juga meminimalkan kemungkinan kesalahan karena pengambilan data dilakukan secara otomatis. Lebih lanjut, database juga dapat diperbarui secara otomatis setiap kali proses scraping dilakukan, sehingga memastikan bahwa data yang tersedia selalu terkini.

Selain menggunakan Go Colly untuk proses scraping, alternatif lain yang dapat dipertimbangkan adalah menggunakan alat lain yang mendukung pembacaan JavaScript. Dengan kemampuan untuk membaca JavaScript, proses filterisasi terhadap artikel yang diinginkan dapat dilakukan dengan lebih efisien, seperti menggunakan melakukan filter honeyTrap dan sejenisnya.

DAFTAR PUSTAKA

[1] Munir, Rinaldi. n.d. “Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1)” Informatika. Diakses 24 April 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf

[2] Munir, Rinaldi. n.d. “Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 2)” Informatika. Diakses 24 April 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf

LAMPIRAN

- **Link Repository Github**
  https://github.com/ChaiGans/Tubes2_BarengApin
- **Link Video**
  https://youtu.be/6zM_Xtk5yMw