LAPORAN TUGAS BESAR II
IF2211 STRATEGI ALGORITMA

PEMANFAATAN ALGORITMA IDS DAN BFS DALAM PERMAINAN WIKIRACE

Kelompok 25 “PathFinder”

Anggota :

13522047 Farel Winalda

13522077 Enrique Yanuar

13522113 William Glory Henderson

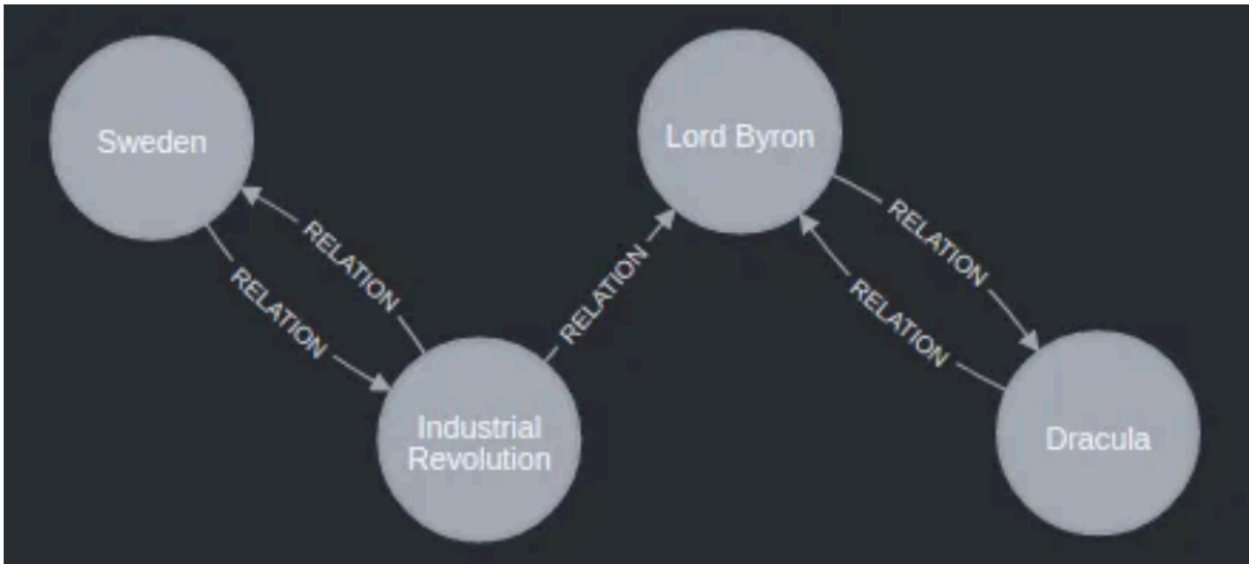
PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG 2023

BAB I
DESKRIPSI TUGAS

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



Gambar 1.1 Ilustrasi Permainan WikiRace

Pada tugas ini, akan diimplementasikan algoritma BFS (*Breadth-First Search*) dan (*Iterative Deepening Search*) untuk menentukan rute terpendek di antara kedua artikel wikipedia dengan aplikasi yang berbasis website dengan algoritma dalam bahasa pemrograman Go. Adapun kakas pembuatan aplikasi berbasis website yang akan digunakan meliputi *React* yang merupakan *framework* dari bahasa pemrograman Javascript.

BAB II
LANDASAN TEORI

A. Penjelajahan Graf

Graf secara matematis dapat didefinisikan sebagai suatu cara yang efektif untuk merepresentasikan kumpulan dari objek-objek dan hubungan antar mereka. Suatu graf memiliki 2 komponen utama yaitu simpul (*vertex*) dan sisi (*edges*). Suatu graf seringkali didefinisikan sebagai pasangan berurutan dengan persamaan matematis $G = (V, E)$, dimana V mewakili himpunan simpul yang terdapat pada graf dan E mewakili himpunan sisi yang menghubungkan pasangan simpul.

Penjelajahan graf merupakan sebuah metode atau algoritma yang mengunjungi atau menelusuri simpul-simpul di dalam graf. Secara umum, terdapat 2 metode yang paling sering digunakan untuk melakukan penjelajahan graf yaitu metode pencarian secara melebar (BFS atau *Breadth-First Search*) dan metode pencarian secara mendalam (DFS atau *Depth-First Search*).

Dalam konteks algoritma pencarian solusi berbasis graf, algoritma ini dapat dikategorikan menjadi dua jenis, yaitu pencarian tanpa informasi (*uninformed* atau *blind search*) dan pencarian dengan informasi (*informed search*). Pencarian tanpa informasi seperti DFS, BFS, DLS (*Depth Limited Search*), IDS (*Iterative Deepening Search*), dan *Uniform Cost Search* merupakan metode pencarian yang hanya mengandalkan struktur dari graf yang ada tanpa menggunakan data tambahan. Di sisi lain, pencarian dengan informasi menggunakan pendekatan heuristik untuk membantu memfokuskan pencarian ke arah yang lebih mungkin untuk menemukan solusi. Algoritma ini mencakup algoritma *Best First Search* dan A*.

Selain itu, dalam proses pencarian solusi terdapat dua pendekatan terhadap representasi graf yaitu graf statis dan graf dinamis. Graf statis merupakan graf yang strukturnya sudah terbentuk sebelum proses pencarian dilakukan. Sedangkan graf dinamis merupakan graf yang strukturnya terbentuk saat proses pencarian dilakukan.

B. Algoritma BFS (*Breadth First Search*)

Breadth-First Search (BFS) adalah algoritma penjelajahan graf yang memulai proses pencarian dari simpul akar dan mengeksplorasi semua tetangga dari simpul tersebut sebelum

melanjutkan ke tetangga berikutnya dan menggunakan antrian untuk menjaga urutan penjelajahan. Algoritma ini menggunakan strategi untuk mencari secara melebar terlebih dahulu. Algoritma ini menggunakan struktur data antrian untuk mengatur simpul yang harus dikunjungi dan untuk menjaga urutan penjelajahan sesuai dengan level kedalaman dari simpul akar.

Pada implementasinya, BFS memulai dengan memasukkan simpul akar ke dalam antrian. Selama antrian tidak kosong, simpul yang berada di depan antrian diambil dan ditandai bahwa simpul sudah dikunjungi. Selanjutnya, semua tetangga dari simpul ini yang belum dikunjungi dimasukkan ke dalam antrian, dan proses ini terus berulang. BFS ini sangat efektif untuk mencari jalur terpendek dalam graf yang tidak berbobot. BFS juga digunakan dalam algoritma yang lebih kompleks seperti mencari komponen terhubung dalam graf atau dalam algoritma yang berhubungan dengan pencarian struktur pohon minimum.

C. Algoritma IDS (*Iterative Deepening Search*)

Iterative Deepening Search (IDS) merupakan algoritma pencarian yang menggabungkan keefektifan BFS dalam pencarian lebar dan kekuatan DFS dalam pencarian mendalam. IDS bekerja seperti DFS tetapi dengan batasan kedalaman yang meningkat secara bertahap. Pada dasarnya, IDS menjalankan DFS berulang kali dengan kedalaman yang dibatasi (*Depth Limited Search* atau DLS) dan meningkatkan batas kedalaman tersebut setelah setiap iterasi lengkap melalui graf. Pendekatan ini menghindari risiko DFS yang terperangkap dalam kedalaman graf yang tak terbatas, sambil membatasi konsumsi memori yang lebih besar dari DFS konvensional.

Iterative Deepening Search sangat ideal untuk pencarian di ruang status yang besar dan tidak diketahui karena menyediakan solusi yang optimal dengan penggunaan memori yang efisien. Pada setiap iterasi, pencarian dilakukan hingga kedalaman tertentu, dan jika solusi tidak ditemukan, maka kedalaman itu ditingkatkan. Keuntungan utama dari IDS adalah menggabungkan manfaat dari BFS dalam menemukan solusi dengan kedalaman minimum, sementara masih mempertahankan kebutuhan memori yang rendah dari DFS. IDS khususnya berguna dalam situasi di mana kedalaman solusi tidak diketahui sebelumnya, membuat pendekatan ini lebih fleksibel.

D. Aplikasi Website yang Dikembangkan

Aplikasi berbasis website yang dikembangkan akan mengimplementasikan pencarian solusi berbasis BFS (*Breadth-First Search*) dan IDS (*Iterative Deepening Search*) untuk

menentukan rute terpendek yang diperlukan dari artikel asal menuju artikel tujuan. Rute ini berdasarkan banyaknya jumlah hipertaut yang ditempuh dari artikel asal ke tujuan. Proses pengumpulan data hipertaut yang terdapat pada artikel Wikipedia tersebut menggunakan teknik *web scraping.*

Web scraping adalah teknik untuk mengekstraksi data dari sebuah halaman *web* secara otomatis dengan menggunakan bot atau program komputer. Teknik ini digunakan pengguna untuk mengumpulkan informasi yang terdapat dari kode HTML pada berbagai situs web tanpa perlu melakukan pengumpulan data secara manual. Dalam proses *web scraping*, halaman web diakses dengan menggunakan HTTP atau protokol lainnya dan kemudian dianalisis untuk mengekstrak data yang diinginkan.

Salah satu *library* yang dapat digunakan untuk melakukan *web scraping* dalam bahasa *Go* adalah *GoColly*. *GoColly* menyediakan berbagai fitur yang berguna untuk mengikuti tautan (*link following*), menangani form, mengabaikan atau memfilter elemen tertentu, dan lain-lain. Selain itu, *GoColly* juga mendukung penggunaan *Goroutine* untuk meningkatkan efisiensi dan kecepatan dalam melakukan scraping pada skala besar.

Pada tugas besar ini, akan diimplementasikan sebuah website untuk menentukan rute terpendek dari dua artikel Wikipedia dengan menggunakan sistem *Frontend* dan *Backend*. Pada *Frontend,* digunakan library *Next JS* yang merupakan salah satu *framework* bahasa pemrograman *JavaScript* yang populer untuk membangun antarmuka pengguna yang responsif dan interaktif. Untuk beberapa komponen juga memanfaatkan *library* dari *NextUI*. Sementara itu, untuk bagian *Backend*, digunakan bahasa pemrograman *Go* dan *Gin* serta *library GoColly* untuk memproses data hipertaut dari artikel Wikipedia dengan algoritma BFS dan IDS untuk menentukan rute terpendek dari kedua artikel yang disediakan. Selain itu juga terdapat database *PostgreSQL* yang digunakan untuk menyimpan hasil scraping yang dapat mempercepat dalam menemukan rute.

BAB III
ANALISIS DAN PEMECAHAN MASALAH

A. Langkah - Langkah Pemecahan Masalah

1. Pendefinisian Masalah

Website yang dikembangkan mempunyai tujuan untuk memungkinkan pengguna dalam mencari jalur terpendek yang menghubungkan dua halaman Wikipedia dengan menerima masukan halaman sumber dan tujuan dari pengguna. Pengguna dapat memilih untuk menggunakan algoritma Breadth-First-Search (BFS) atau Iterative Deepening Search (IDS) dalam melakukan pencarian. Tujuan utama dari *website* ini adalah untuk memberikan solusi terpendek yang tepat dan cepat serta akurat. *Website* ini akan menampilkan jalur (simpul) yang ditemukan dalam bentuk graf.

2. Menetapkan Desain Solusi

***F**ramework frontend* yang digunakan *framework Next JS* dan menggunakan *styling Tailwind CSS*. Untuk *framework backend* akan digunakan *framework Gin* dengan bahasa *Go* serta memanfaatkan *library GoColly* untuk *web scraping*. Website juga menggunakan database *PostgreSQL* untuk menyimpan hasil scraping. Untuk tampilan desain (*UI/UX*) memanfaatkan *library NextUI*. Bentuk tampilan website akan dibagi menjadi 3 page yaitu *Home*, *App*, dan *About*. *Page home* akan berisi penjelasan singkat mengenai website yang dibuat. *Page app* akan ada suatu *form* yang berisi *button switch* untuk memilih BFS atau IDS serta *button switch* untuk memilih ingin hasilnya *single path* atau *multi path*. Selain itu, terdapat tempat untuk memasukkan artikel sumber dan tujuan serta dapat menukar posisi artikel tersebut. Pada tempat ini tersedia juga *dropdown suggestion*. Kemudian ada tombol GO untuk memulai pencarian dan setelah selesai akan menampilkan path dalam bentuk graf dengan menggunakan *library d3.js*.

3. Integrasi dengan Backend

Saat *user* menekan tombol GO akan dilakukan *Fetch API* yang digunakan untuk mengirimkan permintaan dari *frontend* ke *backend*. Data yang dikirim berupa url halaman

asal, url halaman tujuan, algoritma yang dipilih, dan pilihan jenis pathnya. Di backend akan terdapat *API* endpoint yang akan menerima request dari frontend dan kemudian menerima data yang dikirim, lalu memproses data tersebut berdasarkan algoritma yang dipilih.

4. Pemrosesan dan Pengoptimalan

Data yang diterima akan diproses dengan algoritma pencarian yang dipilih. Proses *scraping* dimulai dari artikel asal. Hasil *scraping* akan dimasukkan ke dalam array yang isinya string. Lalu dari hasil *scraping* ini akan dilakukan pengecekan apakah ada yang sesuai dengan artikel tujuan. Jika tidak ada, maka akan dilakukan *scraping* kembali dari array yang berisi hasil *scraping* dari artikel asal. Proses ini dilakukan berulang-ulang hingga ditemukan artikel tujuan. Proses pencarian ini juga menerapkan pengoptimalan dengan menggunakan beberapa Goroutine (worker) untuk melakukan pencarian secara paralel dan konkuren. Selain itu, kode ini juga menyimpan hasil *scraping* ke dalam database. Sehingga ketika ingin melakukan pencarian kembali, hasil akan lebih cepat ditemukan karena tidak perlu melakukan *scraping* ulang. Dengan menerapkan Goroutine dan pemanfaatan database ini diharapkan dapat menemukan solusi dengan lebih cepat.

5. Respon ke Frontend

Hasil dari pencarian akan dikirim ke frontend dan ditampilkan dalam bentuk graf yang memiliki animasi. Graf tersebut menampilkan artikel awal, artikel tujuan, dan artikel-artikel yang dilaluinya. Graf tersebut dapat digerakkan setiap simpulnya, dapat digeser, dan dapat diperbesar atau diperkecil. Node pada graf juga dapat ditekan dan akan mengarahkan langsung ke web wikipedia node tersebut. Selain itu, juga terdapat tulisan mengenai waktu eksekusi, jumlah artikel yang *discraping*, dan kedalaman atau derajat dari algoritma BFS atau IDS. Graf ini memanfaatkan *library d3.js.* Selain itu, terdapat hasil pathnya juga dalam bentuk card seperti pada web *six degrees of wikipedia*.

B. Pemetaan Masalah menjadi Elemen-elemen Algoritma BFS dan IDS

1. Pemetaan menjadi Elemen Algoritma BFS

Elemen:

- String Awal : Url Artikel Wikipedia awal.
- String Tujuan : Url Artikel Wikipedia tujuan.
- Tujuan : Menemukan jalur terpendek dari artikel awal ke artikel tujuan dengan menelusuri link-link yang ada pada setiap artikel secara melebar baru mendalam
- Type struct data : Menyimpan data mengenai link suatu url dan parent (keluarga dari paling atas sampai url tersebut)
- Array of data (urls) : Menyimpan artikel yang akan di *scraping*
- Array of data (newUrls) : Menyimpan seluruh artikel hasil *scraping*
- Map of string boolean (check) : Mencatat semua artikel yang sudah dikunjungi untuk meminimalisir jumlah *scraping*

Proses:

1. Masukkan hasil *scraping* dari url artikel asal ke array of data (urls) sambil dilakukan pengecekan apakah target ditemukan atau tidak. Jika ditemukan, maka langsung ke langkah 7
2. Lakukan *scraping* dari urls atau load dari database (jika sudah ada) dan hasilnya dimasukkan ke array of data (newUrls) serta dilakukan pengecekan apakah target artikel sudah ditemukan. Hasil *scraping* juga akan dimasukkan ke dalam database
3. Semua url yang sudah dilewati akan disimpan di dalam Map check dengan url (string) sebagai key dan boolean sebagai value. Jika sudah dilewati, maka akan bernilai true
4. Jika belum ketemu, maka data dalam urls akan diubah menjadi data dari newUrls
5. Data dari newUrls akan diperbaharui menjadi kosong kembali dan nantinya akan diisi hasil *scraping* dari urls yang baru
6. Ulangi langkah 2 sampai 5
7. Jika sudah ketemu, maka return hasil yang memanfaatkan struct data untuk mengambil seluruh keluarga dari url tersebut

2. Pemetaan menjadi Elemen Algoritma IDS

Elemen:

- String Awal : Url Artikel Wikipedia awal.
- String Tujuan : Url Artikel Wikipedia tujuan.
- Tujuan : Menemukan jalur terpendek dari artikel awal ke artikel tujuan dengan menelusuri link-link yang ada pada setiap artikel secara mendalam yang dibatasi oleh kedalaman yang terus bertambah jika seluruh jalur sudah ditelusuri
- Depth Limit : Batasan kedalaman yang bertambah secara iteratif (batasnya 6).
- Type struct graph : Menyimpan map of string array of string dan mutex. Keynya adalah url dan array of string adalah hasil *scrapingnya*. Mutex untuk melakukan lock saat memasukkan data karena multithread
- Array of string (urls) : Menyimpan url artikel yang akan di *scraping*
- Array of string (newUrls) : Menyimpan url seluruh artikel hasil *scraping*
- Map of string boolean (check) : Mencatat semua artikel yang sudah dikunjungi untuk meminimalisir jumlah *scraping* (saat *scraping*)
- Map of string boolean (visited) : Mencatat semua artikel yang sudah dikunjungi untuk meminimalisir jumlah *scraping* (saat melakukan traversal atau pencarian)
- Array of string (path) : Menyimpan seluruh url artikel yang sedang dilalui (cara kerjanya menyerupai stack)
- Array of array of string (allPaths) : Menyimpan hasil copy dari data path jika path mencapai url target

Proses:

1. Masukkan url artikel awal ke dalam array of string (urls)
2. Lakukan *scraping* dari urls atau load dari database (jika sudah ada) dan hasilnya dimasukkan ke array of string (newUrls). Hasil *scraping* juga akan dimasukkan ke dalam database
3. Hasil *scraping* akan dibangun dalam bentuk graf dengan url awal sebagai parent dan hasil *scraping* sebagai anaknya

4. Semua url yang sudah *discraping* akan disimpan di dalam Map check dengan url (string) sebagai key dan boolean sebagai value. Jika sudah *discraping*, maka akan bernilai true
5. Data dalam urls akan diubah menjadi data dari newUrls dan data newUrls akan dikosongkan
6. Lakukan pencarian pada graf secara dls yang cara kerjanya mirip dengan dfs tetapi dibatasi oleh kedalaman yang nantinya akan bertambah jika seluruh jalur pada kedalaman tersebut tidak ditemukan url tujuan
7. Dalam algoritma dls, ada array of string (path) yang berguna untuk menyimpan jalur yang sedang ditempuh pada kondisi saat itu juga. Ketika melakukan traversal, elemen terakhir dari path akan dicek sudah mencapai target atau belum. Jika sudah mencapai tujuan, maka semua elemen akan dicopy ke dalam array of array of string (allPaths). Jika belum, maka akan dilakukan backtracking dengan cara melakukan pop pada path.
8. Dalam proses traversal, semua url yang sudah dilewati oleh path akan disimpan di dalam Map visited dengan url (string) sebagai key dan boolean sebagai value. Jika sudah dilewati, maka akan bernilai true. Ketika dilakukan backtracking, maka elemen path yang bersangkutan akan dijadikan false pada map visited
9. Jika selama proses dls belum ditemukan, maka akan lanjut ke kedalaman berikutnya dengan mengulangi langkah 2 sampai 8
10. Jika ditemukan, maka return hasil allPaths

C. Fitur Fungsional dan Arsitektur Aplikasi

1. Fitur Fungsional

Ada beberapa fitur fungsional yang terdapat pada aplikasi website ini, antara lain:

- Tombol switch untuk memilih antara algoritma BFS atau IDS
- Terdapat Input box untuk artikel source dan destination yang disertai dengan fitur dropdown rekomendasi untuk memudahkan user dalam memilih judul page yang ada di wikipedia. Dropdown rekomendasi ini akan berubah setiap pengguna melakukan pengetikan per huruf.

- Tombol switch untuk source dan destination sehingga user dapat melakukan pertukaran secara langsung antara source dan destination.
- Tombol switch untuk menentukan hasilnya ingin single path atau multi path
- Tombol GO untuk mengirimkan request ke backend untuk mencari hasil jalur terpendek dari judul awal ke judul tujuan.
- Grafik responsif untuk mengilustrasikan hasil jalur terpendek dari page wikipedia awal ke page wikipedia tujuan. Grafik tersebut dapat digerakkan simpulnya, dapat digeser grafnya, dan dapat diperbesar atau diperkecil
- Card Wikipedia untuk menampilkan hasil jalur terpendek dari page wikipedia awal ke page wikipedia tujuan. Card ini dapat diklik untuk langsung membuka ke link url wikipedia yang bersangkutan.

2. Arsitektur Aplikasi

```
.
├── src/
│   ├── be/
│   │   ├── config/
│   │   │   ├── config.go
│   │   │   ├── config.prod.yaml
│   │   │   └── config.yaml
│   │   ├── controller/
│   │   │   ├── bfs.go
│   │   │   └── ids.go
│   │   ├── database/
│   │   │   ├── queries/
│   │   │   │   └── query.sql
│   │   │   ├── connection.go
│   │   │   └── schema.sql
│   │   ├── repository/
│   │   │   ├── db.go
```

```
│   │   │   ├── models.go
│   │   │   └── query.sql.go
│   │   ├── routes/
│   │   │   └── api.go
│   │   ├── schema/
│   │   │   ├── data_sc.go
│   │   │   └── input_sc.go
│   │   ├── service/
│   │   │   └── s_colly.go
│   │   ├── utilities/
│   │   │   └── scraping.go
│   │   ├── arial.ttf
│   │   ├── Dockerfile
│   │   ├── go.mod
│   │   ├── go.sum
│   │   └── main.go
│   └── fe/
│       ├── app/
│       │   ├── about/
│       │   │   ├── layout.tsx
│       │   │   └── page.tsx
│       │   ├── app/
│       │   │   ├── layout.tsx
│       │   │   └── page.tsx
│       │   ├── error.tsx
│       │   ├── layout.tsx
│       │   ├── page.tsx
│       │   └── providers.tsx
│       ├── components/
│       │   ├── button.tsx
│       │   ├── card.tsx
```

```
│       │   ├── counter.tsx
│       │   ├── Graph.tsx
│       │   ├── iconinfinity.tsx
│       │   ├── icon.tsx
│       │   ├── inputsearch.tsx
│       │   ├── loading.tsx
│       │   ├── navbar.tsx
│       │   ├── particles.tsx
│       │   ├── primitives.ts
│       │   ├── switch.tsx
│       │   ├── theme-switch.tsx
│       │   └── wikicard.tsx
│       ├── config/
│       │   ├── fonts.ts
│       │   └── site.ts
│       ├── public/
│       │   ├── img/
│       │   │   ├── tes.png
│       │   │   ├── wikipedia.png
│       │   │   ├── enrique.jpg
│       │   │   ├── farel.jpg
│       │   │   ├── foto.jpg
│       │   │   └── wg.jpg
│       │   ├── favicon.ico
│       │   ├── logo.svg
│       │   ├── next.svg
│       │   ├── swap.svg
│       │   └── vercel.svg
│       ├── styles/
│       │   └── globals.css
│       ├── types/
```

```
│       │   └── index.ts
│       ├── Dockerfile
│       ├── next.config.js
│       ├── package-lock.json
│       ├── package.json
│       ├── postcss.config.js
│       ├── tailwind.config.js
│       └── tsconfig.json
├── doc/
│   └── PathFinder.pdf
├── docker-compose.yml
└── README.md
```

Untuk aplikasi ini, kami menggunakan React Js + Tailwind CSS untuk frontend nya , kemudian untuk kami menggunakan framework GIN dengan bahasa pemrograman Go. Dalam root project kami terdapat folder src yang didalamnya terdapat dua folder frontend dan backend. Kami menggabungkan frontend dan backend dalam satu repository. Selain itu, di root project juga terdapat readme file yang berisi tata cara menjalankan aplikasi , spesifikasi yang dibutuhkan untuk menjalankan aplikasi dan developer yang mengembangkan aplikasi ini.

Terdapat beberapa program yang dipisah ke dalam beberapa bagian dalam bentuk folder dan juga file dalam pembuatan program server side / backend, antara lain sebagai berikut:

- Folder *config*: Folder ini berisi program untuk *settings* / pengaturan dari database yang dibuat seperti host, port, password, dan username.
- Folder *controller*: Folder ini berisi fungsi-fungsi dari beberapa route yang dibagi berdasarkan endpoint *bfs* dan *ids* / metode pencarian *bfs* dan *ids*, beserta response dan status yang dikembalikan.
- Folder *routes*: Folder ini berisi implementasi dari pembuatan endpoint dari *api*, method (*Post*) dari endpoint, dan juga pengaturan *CORS* (Cross Origin Method Sharing).

- Folder *schema*: Folder ini berisi struktur dari *Body Request* yang diterima dan divalidasi dari request aplikasi *Next* (frontend) dan juga struktur data lain yang menunjukkan *parent url* dari suatu *url*.
- Folder *service*: Folder ini berisi pengaturan yang digunakan dalam implementasi dari library *colly* untuk *scraping* dari *url wikipedia*.
- Folder *utilities*: Folder ini berisikan fungsi-fungsi yang diekspor dalam melakukan *scraping url wikipedia*.
- File *Dockerfile*: File yang berisi pengaturan dan perintah yang dapat dipanggil oleh pengguna dalam pembuatan membangun Image dalam Container yang telah disediakan.
- File *go.mod & go.sum*: File ini berisi beberapa package dari *library* dan *dependencies* yang digunakan dalam logika-logika backend dan implementasi program.
- File *main.go*: File ini merupakan file utama yang memanggil seluruh fungsi-fungsi untuk mengatur *port*, *endpoint*, *method*, serta *scraping*.
- Folder *database*: Folder ini berisikan *pool connection* ke database dan berisi query sql yang akan di *generate* menjadi fungsi dalam Golang menggunakan *sqlc.*
- Folder *repository*: Folder ini berisikan hasil *generate* dari query yang digenerate oleh *sqlc*
- File *sqlc.yaml*: File ini berisikan konfigurasi ke database dan mengatur *directory sql* yang akan di *generate* menjadi file Golang

Dalam pembuatan frontend menggunakan *NextJs* dan *NextUI*, implementasi program dibagi menjadi beberapa folder dan file, sebagai berikut:

- Folder *app*: Folder ini berisi homepage, layout page, error page, dan page-page yang dipisah lagi menjadi beberapa folder sesuai dengan dokumentasi yang diberikan oleh *Next* routing.
- Folder *components*: Folder ini berisi beberapa komponen-komponen UI yang digunakan dalam page yang berupa potongan program yang diekspor, maupun logo-logo / icon yang juga diekspor ke dalam page utama.
- Folder *config*: Folder ini berisi tentang pengaturan tampilan, seperti font yang digunakan dan juga beberapa isi dari komponen Navbar beserta metadata (nama, deskripsi) dari aplikasi *Next*.

- Folder *public*: Folder ini menyimpan beberapa gambar dan asset dengan format tertentu yang digunakan dalam pembuatan page utama
- Folder *style*: Folder ini berisi file css yang digunakan untuk membuat styling tanpa menggunakan syntax dari tailwind.
- Folder *types*: Folder ini berisi tipe-tipe interface maupun struktur data yang diperlukan dalam pembuatan page utama.
- File *DockerFile*: File yang berisi pengaturan dan perintah yang dapat dipanggil oleh pengguna dalam pembuatan membangun Image dalam Container yang telah disediakan.
- File *next.config.js*: File ini berisi pengaturan modul untuk men-*developing* framework dari *Next*.
- File *package.json* & *package-lock.json*: File ini berisi library-library dan *dependencies* yang diperlukan dan akan di-*install* ketika menjalankan '*npm install*' dalam menjalankan aplikasi *Next*.
- File *postcss.config.js*: File ini berisi *plugin* untuk *tailwind* dan beberapa konfigurasi / pengaturan css dalam aplikasi *Next*.
- File *tailwind.config.js*: File konfigurasi untuk Tailwind CSS, sebuah framework styling yang digunakan untuk mendesain tampilan aplikasi dengan efisien.
- File *tsconfig.json*: File ini berisi konfigurasi dari *Typescript* (Bahasa pemrograman yang digunakan dalam aplikasi *Next* yang kami pakai), seperti versi dan modul-modul *Typescript* yang digunakan.

D. Contoh Ilustrasi Kasus

Salah satu contoh kasus yang dapat diselesaikan dengan program berbasis website yang mengimplementasikan algoritma BFS dan IDS adalah menentukan jumlah link atau hipertaut yang perlu diklik dari artikel Wikipedia dengan judul Joko Widodo (en.wikipedia.org/wiki/Joko_Widodo) menuju artikel Wikipedia dengan judul Indonesia(en.wikipedia.org/wiki/Indonesia). Hasilnya kedalamannya adalah 2 derajat.

Dengan algoritma BFS, artikel asal akan dilakukan scraping dan hasilnya disimpan ke dalam array of string (array 1) yang cara kerjanya mirip queue. Hasil array akan diiterasi dan dicek apakah memiliki artikel tujuan atau tidak. Semua url yang telah dilewati akan ditandai agar tidak terjadi perulangan. Jika tidak ada, maka array tersebut akan dilakukan scraping untuk setiap

elemennya dan hasilnya disimpan ke dalam array of string yang berbeda (array 2). Jika masih belum, maka data dalam array 1 akan diubah menjadi data dari array 2 dan array 2 akan dikosongkan. Kemudian array 1 akan di scraping lagi seperti langkah awal dan hal ini akan dilakukan berulang-ulang sampai mendapatkan artikel tujuan. Untuk single path, jika sudah ketemu satu jawaban maka program akan di stop. Sedangkan untuk multi path, program akan terus berlanjut sampai semua artikel pada kedalaman tersebut selesai ditelusuri.

Dengan algoritma IDS, artikel asal akan disimpan dalam array 1 dan digunakan sebagai parent dari graf untuk membangun node-node (artikel hasil scraping). Hasil scraping ini akan disimpan dalam array 2. Data dalam array 1 akan diubah menjadi data dari array 2. Data dari array 2 akan diperbaharui menjadi kosong kembali dan nantinya akan diisi hasil scraping dari array 1 yang baru. Setelah selesai pembangunan graf dari kedalaman 1, akan dilakukan pencarian secara DLS yang cara kerjanya mirip DFS tetapi dibatasi oleh suatu kedalaman. Dalam proses DLS akan dimanfaatkan array yang cara kerjanya seperti stack. Array tersebut akan menyimpan seluruh path atau jalur yang sedang ditempuh pada saat itu juga (dari parent sampai node pada kedalaman terakhir). Ketika sudah mencapai ujung dan tidak ditemukan, maka akan dilakukan backtracking dan array akan di pop setiap kali melakukan backtrack. Pada saat menempuh jalur, path yang dilalui akan ditandai true agar tidak dapat terjadi pengulangan dan ketika backtracking path tersebut, maka path tersebut akan ditandai menjadi false. Ketika sudah menemukan path, maka array tersebut akan dicopy ke array hasil yang nantinya akan di return. Jika tidak ditemukan path di kedalaman tersebut, maka akan dilakukan proses awal lagi yaitu scraping dari array 1 untuk lanjut ke kedalaman berikutnya. Kode akan berulang-ulang sampai menemukan url artikel tujuan. Untuk single path, jika sudah ketemu satu jawaban, maka program akan di stop. Sedangkan untuk multi path, program akan terus berlanjut sampai semua artikel pada kedalaman tersebut selesai ditelusuri.

Semua url hasil scraping dari awal sampai akhir akan disimpan langsung ke database sehingga dapat mempercepat waktu eksekusi. Hal ini dilakukan agar tidak perlu melakukan scraping berulang untuk artikel yang sudah pernah discraping. Sehingga pada saat melakukan scraping akan dilakukan pengecekan terlebih dahulu pada database. Jika sudah ada, maka akan langsung dipakai datanya. Jika tidak, maka perlu discraping terlebih dahulu.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

A. Spesifikasi Teknis Program

1. Implementasi Struktur Data

    a. Algoritma BFS

    Pada algoritma BFS (*Breadth First Search*) diimplementasikan dalam bentuk array of string yang bekerja menyerupai queue. Artikel awal akan dilakukan scraping terlebih dahulu dan hasil scraping akan dimasukkan ke array. Setiap kali melakukan scraping, semua hasil url atau link yang didapatkan akan dicek apakah mencapai url artikel target atau tidak. Jika mencapai, maka pencarian akan diberhentikan (jika single path). Pencarian akan tetap dilanjutkan pada kedalaman tersebut sampai antrian url yang akan discraping hingga semua simpul selesai ditelusuri (jika all path). Jika tidak ditemukan, maka setiap hasil scraping yang sudah di dalam array akan discraping lagi untuk lanjut ke kedalaman berikutnya. Kemudian dilakukan pengecekan kembali apakah sudah menemukan target atau belum. Hal ini akan dilakukan berulang-ulang hingga ketemu artikel tujuan. Jadi pada depth 0 sampai n (ketemu url sesuai target) akan dilakukan scraping secara BFS atau melebar terlebih dahulu baru mendalam. Program ini dioptimasi menggunakan *thread* dan *pool* dari go-colly (untuk scrapping). Batas *thread* dalam program ini adalah 200. Program ini juga menggunakan database sehingga ketika dilakukan pencarian ulang akan ditemukan hasilnya lebih cepat dari yang pertama karena tidak perlu melakukan scraping berulang untuk artikel yang sudah pernah discraping. Untuk penjelasan cara BFS tersedia pada bagian pemetaan masalah.

Struktur data yang dipakai ada beberapa yaitu :

- Type struct data : Menyimpan data mengenai link suatu url dan parent (keluarga dari paling atas sampai url tersebut)
- Array of data (urls) : Menyimpan artikel yang akan di scraping
- Array of data (newUrls) : Menyimpan seluruh artikel hasil scraping
- Map of string boolean (check) : Mencatat semua artikel yang sudah dikunjungi untuk meminimalisir jumlah scraping

```go
package schema

type Data struct {
    Url    string `json:"url"`
    Parent string `json:"parent"`
}
```

```go
package controller

func BfsScrapping(context *gin.Context) {

    var request schema.InputBodyRequest

    if err := context.ShouldBindJSON(&request); err != nil {
        log.Err(err).Msg("Error Bind JSON")
        context.JSON(http.StatusOK, gin.H{"success": false, "message": "Error Bind JSON"})
        return
    }

    validator := validator.New()
    if err := validator.Struct(request); err != nil {
        log.Err(err).Msg("Error Validator")
        context.JSON(http.StatusOK, gin.H{"success": false, "message": "Error Validator"})
        return
    }

    var wg sync.WaitGroup
    var mu sync.Mutex
    var err error
    semaphore := make(chan struct{}, 200)
```

```
    found := false
    check := make(map[string]bool)
    var urls []schema.Data

    // scrap url start
       urls, found, countCompare, err = utilities.ScrapeWikipedia(request.Start, request.Start,
service.Collectors[0], request.End)
    if err != nil {
        log.Err(err).Msg("Error scrap")
        context.JSON(http.StatusOK, gin.H{"success": false})
    }
    service.Data.Store(request.Start, urls)

    result := []schema.Data{}
    if found || request.Start == request.End {
        result = append(result, schema.Data{Parent: request.Start})
    }
    count := 0

    for !found {
        newUrls := []schema.Data{}
        println("ini", len(urls))
        if len(urls) == 0 {
            log.Err(err).Msgf("Error scrap url not found")
            break
        }
        // scrap semua url
        for i, url := range urls {
            if found && !request.IsMulti {
                var resArray [][]string
                for _, p := range result {
                    arr := strings.Split(p.Parent, " ")
                    arr = append(arr, request.End)
                    resArray = append(resArray, arr)
                }
                 context.JSON(http.StatusOK, gin.H{"success": true, "total": count, "total_compare":
countCompare, "result": resArray[0:1]})
                return
            }
            if check[url.Url] {
                continue
            }
            check[url.Url] = true
            count++
            wg.Add(1)
            semaphore <- struct{}{} // Acquire a token
            go func(url schema.Data, i int) {
```

```
                // set collector dari pool untuk thread
                collector := <-service.CollectorPool
                defer wg.Done()
                defer func() { <-semaphore }()
                defer func() { service.CollectorPool <- collector }()

                // mulai logic scrap
                var tempUrls []schema.Data
                var cc int
                var f bool
                   tempUrls, f, cc, err = utilities.ScrapeWikipedia(url.Parent, url.Url, collector,
request.End)
                if err != nil {
                    log.Err(err).Msgf("Error scrap %v", url)
                }

                // update variable dengan lock sebaa=gai pengaman
                mu.Lock()
                if !found {
                    newUrls = append(newUrls, tempUrls...)
                }
                if f {
                    result = append(result, url)
                    found = true
                }
                countCompare += cc
                mu.Unlock()
            }(url, i)
        }

        // untuk memastikan semua thread selesai terlebih dahulu
        wg.Wait()

        urls = newUrls

        if found {
            break
        }
    }

    var resArray [][]string
    for _, p := range result {
        arr := strings.Split(p.Parent, " ")
        if arr[0] != request.End {
            arr = append(arr, request.End)
        }
        resArray = append(resArray, arr)
    }
```

```
    countCompare--

    // in case karena go routine jadi dapat lebih dari 1
    if request.IsMulti {
            context.JSON(http.StatusOK, gin.H{"success": true, "total": count, "total_compare":
countCompare, "result": resArray})
    } else {
            context.JSON(http.StatusOK, gin.H{"success": true, "total": count, "total_compare":
countCompare, "result": resArray[0:1]})
    }

}
```

b. Algoritma IDS

Pada algoritma IDS, digunakan struktur data graph yang diimplementasikan dengan bentuk map dengan key adalah string dan value adalah array of string dan mutex. Dasar dari ide dalam algoritma ini adalah dengan membangun semua node dalam setiap depth terlebih dahulu kemudian akan menggunakan algoritma DLS atau DFS dengan batasan kedalaman untuk melakukan pencarian secara traversal pada graf tersebut dan menentukan apakah target ketemu atau tidak. Untuk ide awalnya sama seperti BFS yaitu memasukkan url artikel awal ke array 1 dan hasil *scrapingnya* dimasukkan ke array 2. Kemudian akan dibangun graf dengan url artikel awal sebagai parent dan hasil *scraping* sebagai anaknya. Selanjutnya dilakukan pencarian DLS yang memanfaatkan array dengan cara kerja yang mirip stack. Pencarian dilakukan secara traversal mendalam terlebih dahulu. Jika untuk 1 jalur sudah mencapai tujuan, maka semua elemen akan dicopy ke dalam array of array of string. Jika sudah mencapai batas kedalaman dan tidak ditemukan url tujuan, maka akan dilakukan backtracking dengan cara melakukan pop pada path. Jika untuk seluruh jalur pada kedalaman tersebut tidak ditemukan url tujuan, maka program akan mengulang proses *scraping* kembali untuk lanjut ke kedalaman berikutnya. *Scraping* yang dilakukan ada pada array 1 yang sudah di set datanya dari array 2 (hasil *scraping*). Untuk single path, jika sudah menemukan, maka akan diberhentikan programnya. Untuk multi path, pencarian akan tetap dilanjutkan

pada kedalaman tersebut sampai antrian url yang akan *discraping* hingga semua simpul selesai ditelusuri.

Struktur data yang dipakai ada beberapa yaitu :

- Type struct graph : Menyimpan map of string array of string dan mutex. Keynya adalah url dan array of string adalah hasil *scrapingnya*. Mutex untuk melakukan lock saat memasukkan data karena program memakai multithread
- Array of string (urls) : Menyimpan url artikel yang akan di *scraping*
- Array of string (newUrls) : Menyimpan url seluruh artikel hasil *scraping*
- Map of string boolean (check) : Mencatat semua artikel yang sudah dikunjungi untuk meminimalisir jumlah *scraping* (saat *scraping*)
- Map of string boolean (visited) : Mencatat semua artikel yang sudah dikunjungi untuk meminimalisir jumlah *scraping* (saat melakukan traversal atau pencarian)
- Array of string (path) : Menyimpan seluruh url artikel yang sedang dilalui (cara kerjanya menyerupai stack)
- Array of array of string (allPaths) : Menyimpan hasil copy dari data path jika path mencapai url target

```
package controller

type Graph struct {
    graph map[string][]string
    mu    sync.RWMutex
}

var found bool
var countCompare int
var count int
var isMulti bool

func NewGraph() *Graph {
    return &Graph{
        graph: make(map[string][]string),
    }
}

func (g *Graph) AddEdge(u, v string) {
    g.mu.Lock()
```

```
    defer g.mu.Unlock()
    if _, ok := g.graph[u]; !ok {
        g.graph[u] = make([]string, 0)
    }
    g.graph[u] = append(g.graph[u], v)
}

func (g *Graph) IDDFS(start string, goal string, maxDepth int) [][]string {
    var allPaths [][]string
    var urls []string
    urls = append(urls, start)
    var wg sync.WaitGroup
    var mu sync.Mutex
    semaphore := make(chan struct{}, 200)
    var err error

    check := make(map[string]bool)
    for depth := 1; depth <= maxDepth; depth++ {
        if found {
            break
        }
        if len(urls) == 0 {
            break
        }

        // scrap all and create edge
        var newUrls []string
        for i, url := range urls {
            if check[url] {
                continue
            }
            check[url] = true
            wg.Add(1)
            semaphore <- struct{}{}
            go func(url string, i int) {
                // set collector dari pool untuk thread
                collector := <-service.CollectorPool
                defer wg.Done()
                defer func() { <-semaphore }()
                defer func() { service.CollectorPool <- collector }()
                var tempUrls []string

                var temp []schema.Data
                temp, _, _, err = utilities.ScrapeWikipedia("", url, collector, goal)
                if err != nil {
                    log.Err(err).Msgf("Error scrap %v", url)
                }
                for _, x := range temp {
```

```
                    tempUrls = append(tempUrls, x.Url)
                }

                for _, x := range tempUrls {
                    g.AddEdge(url, x)
                }

                mu.Lock()
                count++
                newUrls = append(newUrls, tempUrls...)
                mu.Unlock()
            }(url, i)
        }

        wg.Wait()
        urls = newUrls
        visited := make(map[string]bool)
        path := make([]string, 0)
        // mulai dls
        countCompare++
        g.DLS(start, goal, depth, visited, &path, &allPaths)
    }
    return allPaths
}

func (g *Graph) DLS(node string, goal string, depth int, visited map[string]bool, path *[]string,
allPaths *[][]string) {
    g.mu.RLock()
    defer g.mu.RUnlock()

    *path = append(*path, node)
    visited[node] = true

    if node == goal {
        found = true
        newPath := make([]string, len(*path))
        copy(newPath, *path)
        *allPaths = append(*allPaths, newPath)
        // disini???
    } else if depth > 0 {

        for _, neighbor := range g.graph[node] {
            // println("aaaa")
            if found && !isMulti {
                break
            }
            if !visited[neighbor] {
                countCompare++
```

```
                g.DLS(neighbor, goal, depth-1, visited, path, allPaths)
            }
        }
    }

    // Backtrack cuy
    *path = (*path)[:len(*path)-1]
    visited[node] = false
}

func IdsScrapping(context *gin.Context) {
    var request schema.InputBodyRequest

    if err := context.ShouldBindJSON(&request); err != nil {
        log.Err(err).Msg("Error Bind JSON")
        context.JSON(http.StatusOK, gin.H{"success": false, "message": "Error Bind JSON"})
        return
    }

    validator := validator.New()
    if err := validator.Struct(request); err != nil {
        log.Err(err).Msg("Error Validator")
        context.JSON(http.StatusOK, gin.H{"success": false, "message": "Error Validator"})
        return
    }
    g := NewGraph()
    found = false

    start := request.Start
    goal := request.End
    maxDepth := 6
    count = 0
    countCompare = 0
    isMulti = request.IsMulti

    paths := g.IDDFS(start, goal, maxDepth)

    uniqueMap := make(map[string]bool)

    // pastikan result unique
    var uniqueArrayOfArrays [][]string
    for _, arr := range paths {
        arrString := fmt.Sprintf("%v", arr)
        if !uniqueMap[arrString] {
            uniqueMap[arrString] = true
            uniqueArrayOfArrays = append(uniqueArrayOfArrays, arr)
        }
    }
```

```
        context.JSON(http.StatusOK, gin.H{"success": true, "total": count, "total_compare":
countCompare, "result": uniqueArrayOfArrays})
}
```

c. *Scraping*

Ketika program dijalankan, sistem akan membuat pool yang terdiri dari 200 collector menggunakan gocolly. Tujuan dari pembuatan pool ini adalah untuk mendistribusikan beban kerja di antara collector, sehingga mengurangi risiko pemblokiran oleh Wikipedia. Fungsi *scraping* beroperasi dengan menerima URL yang perlu dikunjungi. Sebelum mengakses URL, fungsi ini akan melakukan pengecekan dalam database untuk memastikan data terkait belum ada. Jika data belum tersedia, maka proses *scraping* akan dilakukan. Fungsi ini akan mengambil URL yang mengarah ke halaman Wikipedia lain, kecuali URL yang mengandung tag “display none” atau yang termasuk dalam kelas “nowraplinks” — kelas yang membuat URL tidak dapat diklik langsung dari situs Wikipedia. URL yang juga tidak diambil termasuk yang mengandung “Category:”, “Wikipedia:”, “File:”, “Help:”, “Portal:”, “Special:”, “Talk:”, “User_template:”, “Template_talk:”, “Mainpage:”, dan “Main_Page”. Setelah proses *scraping* selesai, URL dan hasil *scraping* akan disimpan ke dalam database.

```
package utilities

func isExcluded(link string) bool {
    if link == "/wiki/Main_Page" {
        return true
    }
    excludedNamespaces2 := []string{
        "Category:", "Wikipedia:", "File:", "Help:", "Portal:",
        "Special:", "Talk:", "User_template:", "Template_talk:", "Mainpage:", "Main_Page",
    }
    for _, ns := range excludedNamespaces2 {
        if regexp.MustCompile(`^` + regexp.QuoteMeta(ns)).MatchString(link) {
            return true
        }
```

```
    }
    return false
}

func ScrapeWikipedia(parent string, url string, c *colly.Collector, end string) ([]schema.Data,
bool, int, error) {
    // defer wg.Done()
    ctx := context.Background()
    query := repository.New(database.Database)
    uniq := make(map[string]bool)
    found := false

    count := 0

    var foundURLs []schema.Data
    var urls []string
    defer func() {
        foundURLs = nil
        urls = nil
        // uniq = nil
    }()

    // check db
    dataUrls, err := query.GetUrl(ctx, url)
    if err == nil {
        for _, dataurl := range dataUrls.RelatedUrls {
            if dataurl == end && !found {
                    foundURLs = append(foundURLs, schema.Data{Url: dataurl, Parent: parent + " " +
dataurl})
                found = true
                // return
            } else if !uniq[dataurl] {
                count++
                    foundURLs = append(foundURLs, schema.Data{Url: dataurl, Parent: parent + " " +
dataurl})
            }
        }
        return foundURLs, found, count, nil
    }

    // not found do scrap
    combinedRegex := regexp.MustCompile(`^/wiki/([^#:\s]+)$`)

    c.OnHTML("a[href]", func(e *colly.HTMLElement) {
        link := e.Attr("href")
        if combinedRegex.MatchString(link) {
            fullLink := "https://en.wikipedia.org" + link
            if isVisible(e) {
```

```
                if !isExcluded(link) {
                    if fullLink == end && !found {
                         foundURLs = append(foundURLs, schema.Data{Url: fullLink, Parent: parent + "
" + fullLink})
                        found = true
                    } else if !uniq[fullLink] {
                        if !found {
                            count++
                        }
                         foundURLs = append(foundURLs, schema.Data{Url: fullLink, Parent: parent + "
" + fullLink})
                    }
                    urls = append(urls, fullLink)
                }
            }
        }

    })

    // Start the scraping process
    err = c.Visit(url)
    c.Wait()
    if err != nil {
        return nil, false, count, err
    }

    // save to db

    data := repository.SaveUrlParams{
        Url:         url,
        RelatedUrls: urls,
    }
    if urls != nil {
        _ = query.SaveUrl(ctx, data)
    }

    // Return the found Wikipedia URLs
    return foundURLs, found, count, nil
}

func isVisible(e *colly.HTMLElement) bool {
    class := e.Attr("class")
    class = strings.ReplaceAll(class, " ", "")
    if strings.Contains(class, "nowraplinks") {
        return false
    }

    // Check parent elements for visibility
```

```
    for parent := e.DOM.Parent(); parent.Length() != 0; parent = parent.Parent() {
        parentClass, found := parent.Attr("class")
        parentClass = strings.ReplaceAll(parentClass, " ", "")
        if found && strings.Contains(parentClass, "nowraplinks") {
            // fmt.Println(e.Attr("href"))
            return false
        }
    }
    return true
}
```

```
package service

var Collectors [200]*colly.Collector
var CollectorPool chan *colly.Collector
var Data sync.Map

func InitColly(n int) {
    CollectorPool = make(chan *colly.Collector, n)
    for i := 0; i < n; i++ {
        Collectors[i] = colly.NewCollector(
            colly.AllowedDomains("en.wikipedia.org", "www.wikipedia.org"),
            colly.AllowURLRevisit(),
            colly.Async(true),
            colly.CacheDir(""),
        )

          Collectors[i].UserAgent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
        Collectors[i].SetRequestTimeout(15 * time.Second)
        Collectors[i].Limit(&colly.LimitRule{
            DomainGlob:  "*wikipedia.org*", // Adjust according to your target domain
            Parallelism: 5,                 // Number of parallel requests to the domain
        })
        CollectorPool <- Collectors[i] // Add the collector to the pool
    }
}
```

B. Penjelasan Tata Cara Penggunaan Program

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

```
git clone https://github.com/mybajwk/Tubes2_PathFinder.git
```

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

```
cd ./Tubes2_PathFInder
```

3. Setelah pengguna berada pada root directory, jalankan perintah berikut pada terminal

```
docker compose up
```

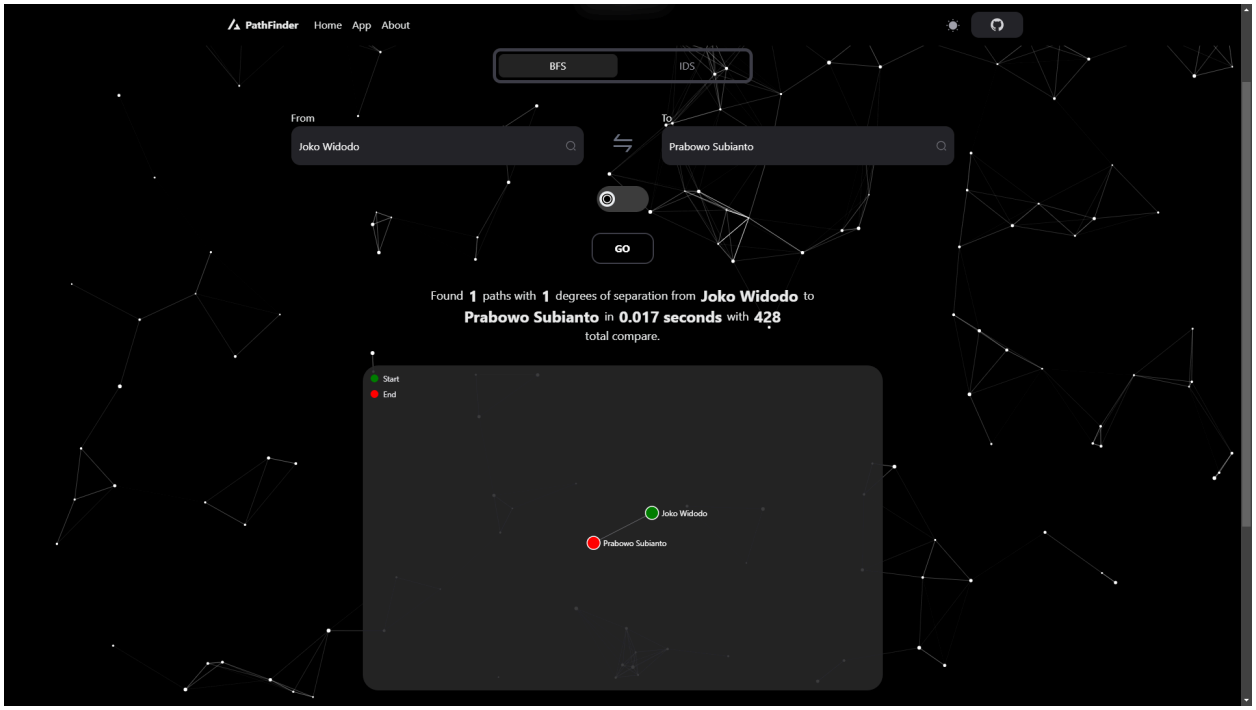
4. Setelah aplikasi berjalan, buka browser web Anda dan kunjungi:

http://localhost:7781

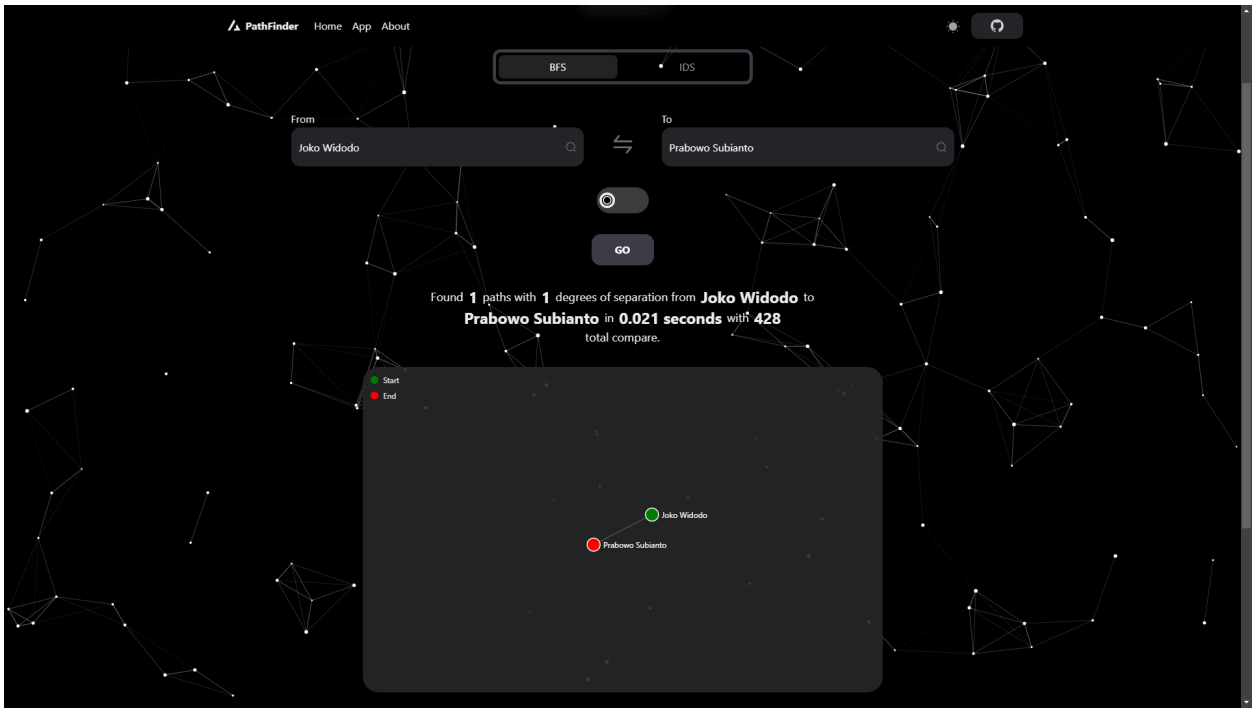
5. Setelah pengguna berhasil membuka website, pengguna dapat memilih algoritma pencarian, baik dengan menggunakan algoritma BFS maupun IDS.
6. Setelah pengguna memilih algoritma pencarian, pengguna menuliskan judul artikel Wikipedia asal dan tujuan, program juga akan memberikan rekomendasi artikel Wikipedia berdasarkan judul yang dimasukkan oleh pengguna.
7. Program akan menampilkan rute terpendek antara kedua artikel dalam bentuk visualisasi graf, wikipedia card, beserta waktu eksekusi, jumlah artikel yang dilalui, beserta kedalaman pencarian.

C. Hasil Pengujian

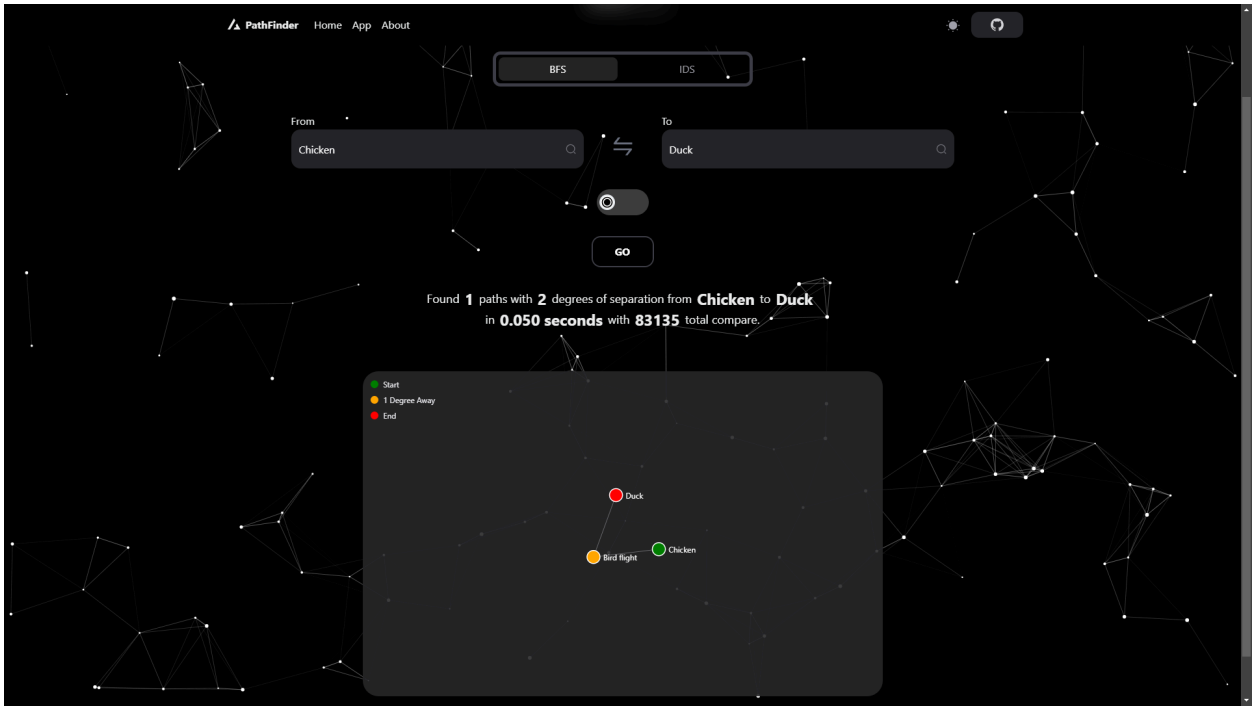
1. Hasil pengujian BFS:
    - BFS *single path* dari *"Jokowi"* menuju *"Prabowo Subianto"* (1 *degree*)



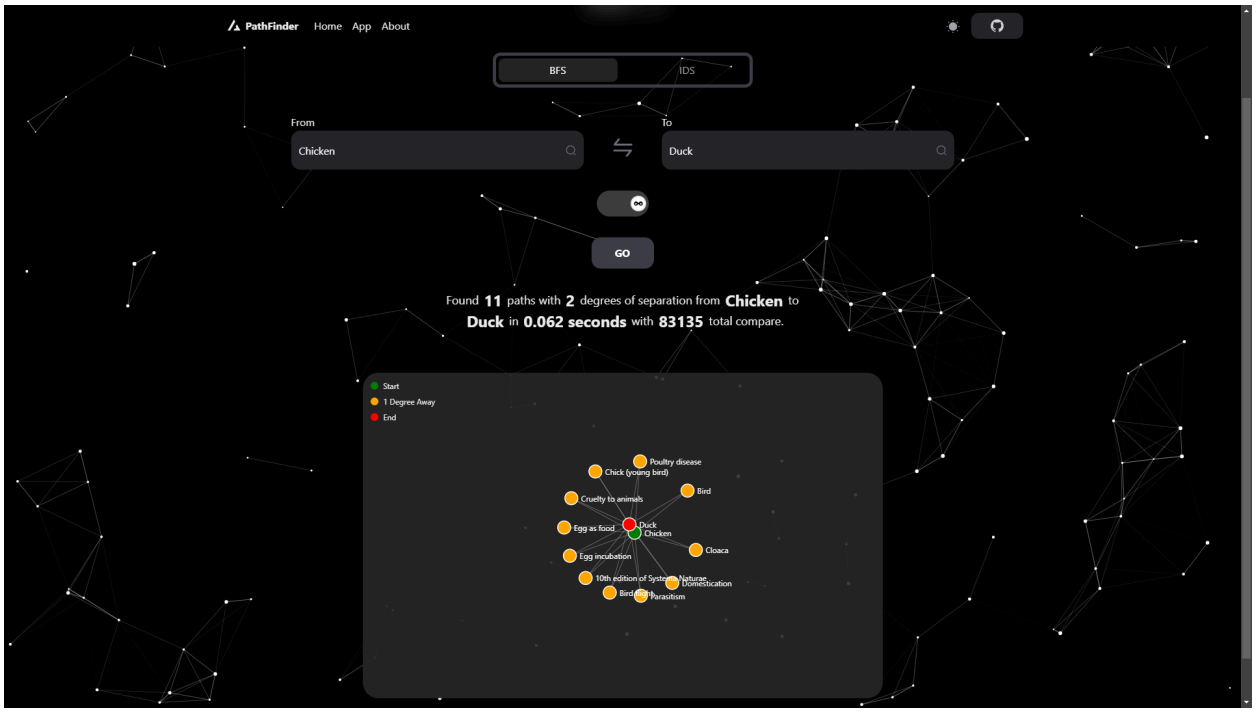
- BFS *multi path* dari *"Jokowi"* menuju *"Prabowo Subianto"* (1 *degree*)



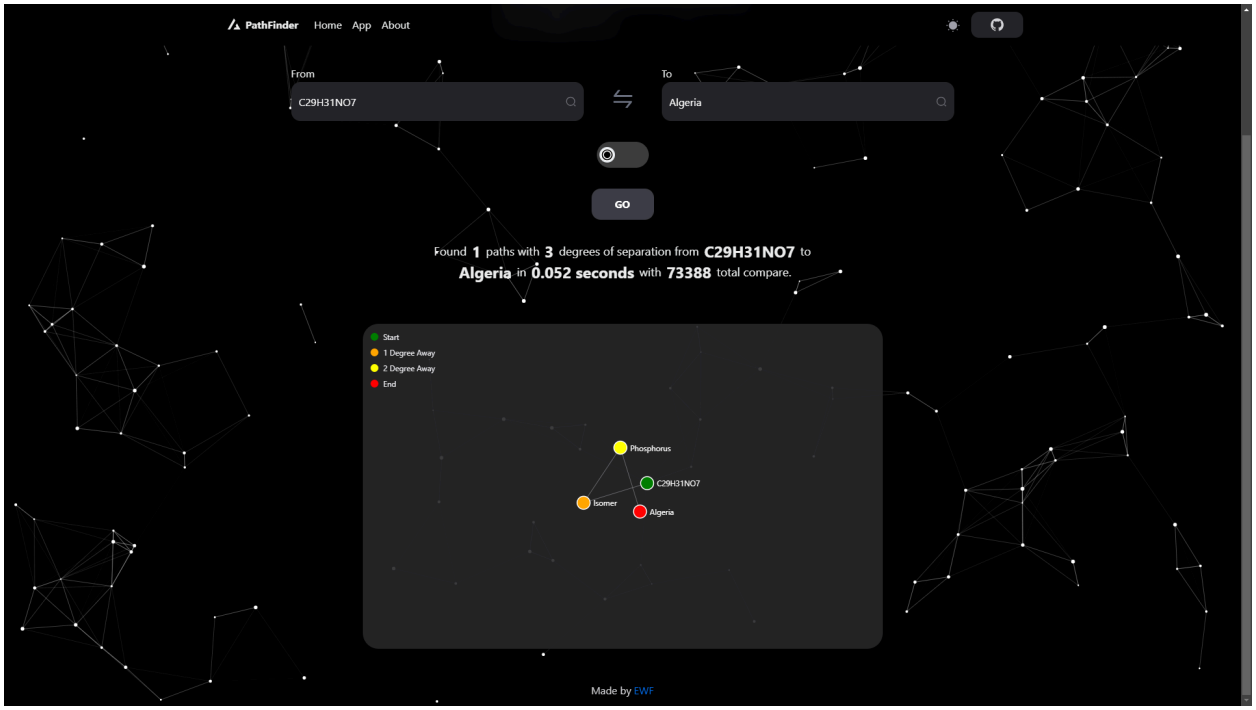
- BFS *single path* dari *"Chicken"* menuju *"Duck"* (2 *degree*)



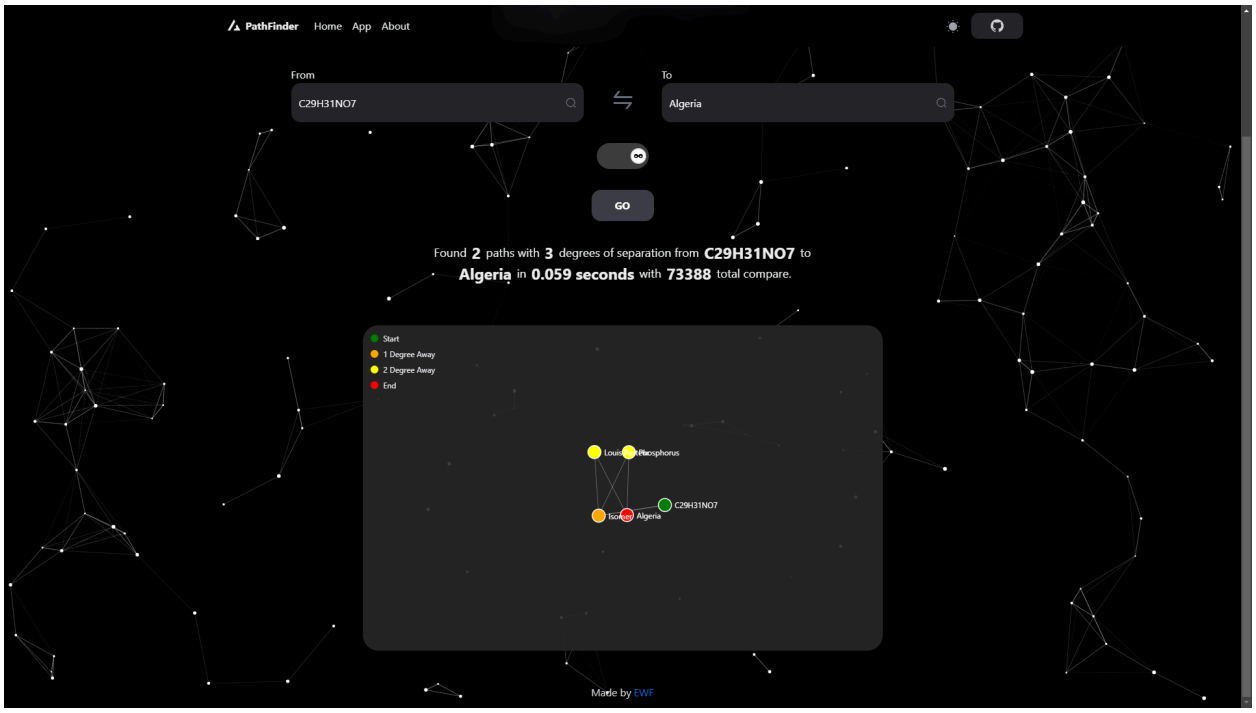
- BFS *multi path* dari *"Chicken"* menuju *"Duck"* (2 *degree*)



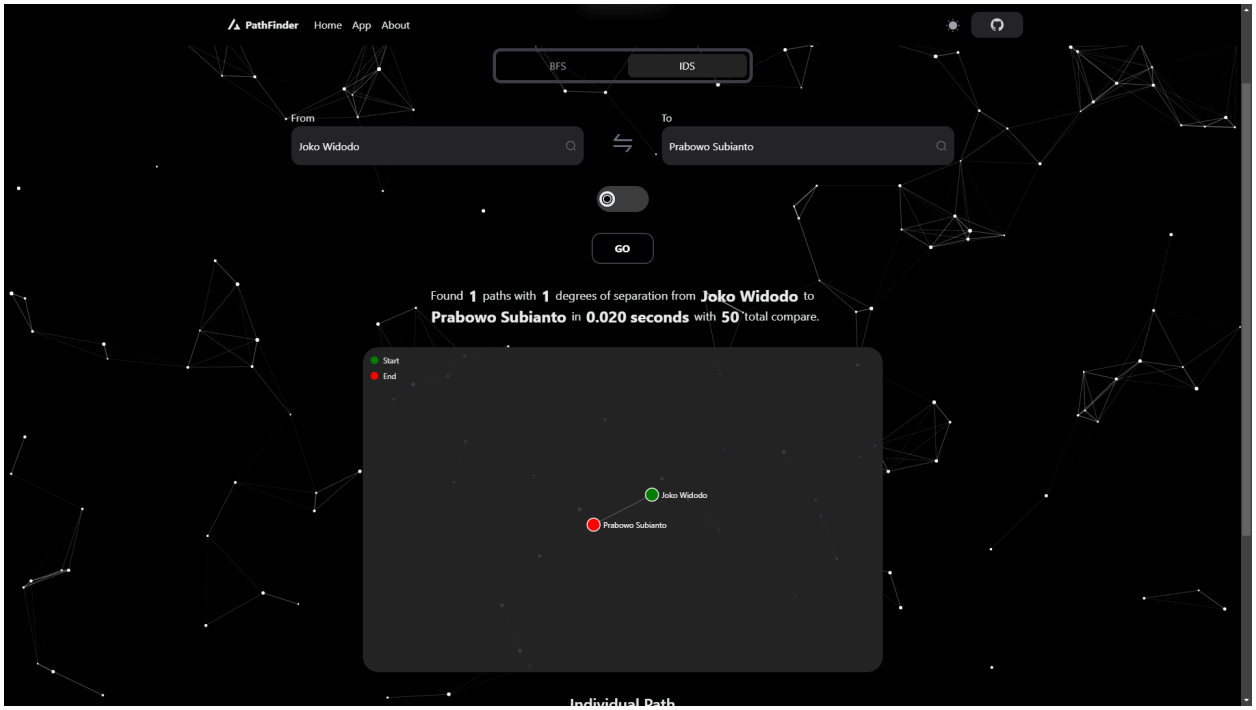
- BFS *single path* dari *"C29H31NO7"* menuju *"Algeria"* (3 *degree*)



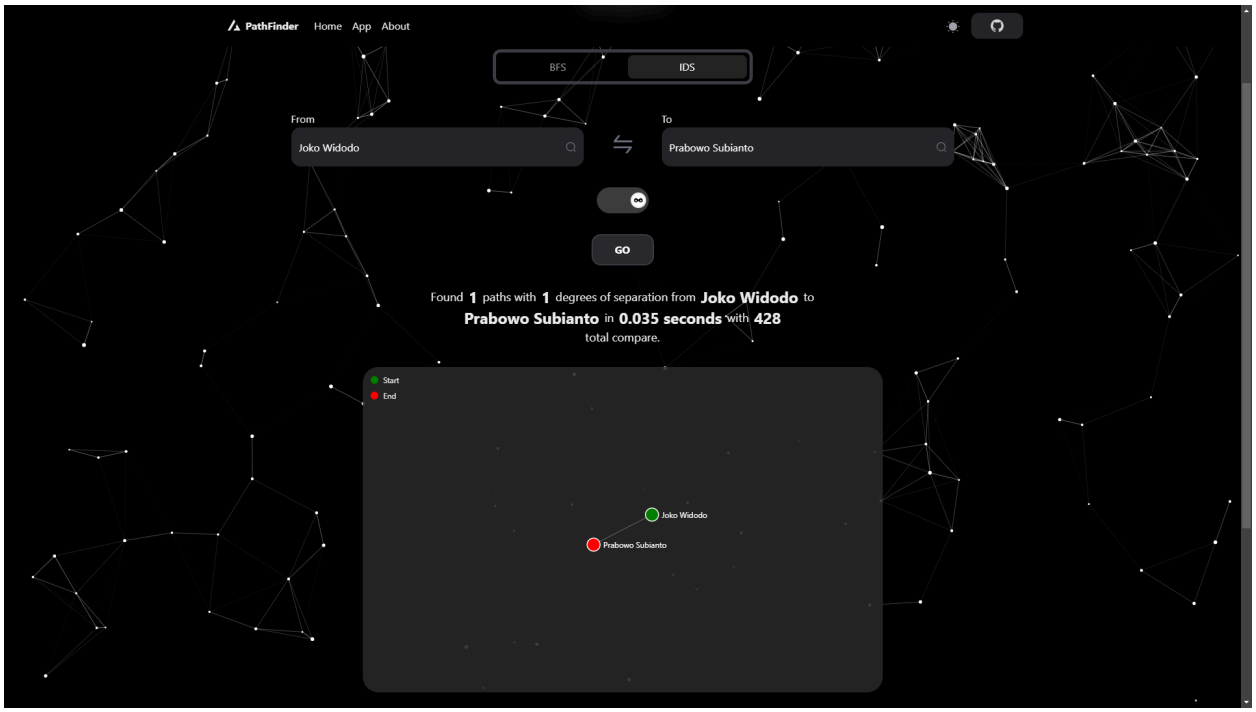
- BFS *multi path* dari *"C29H31NO7"* menuju *"Algeria"* (3 *degree*)



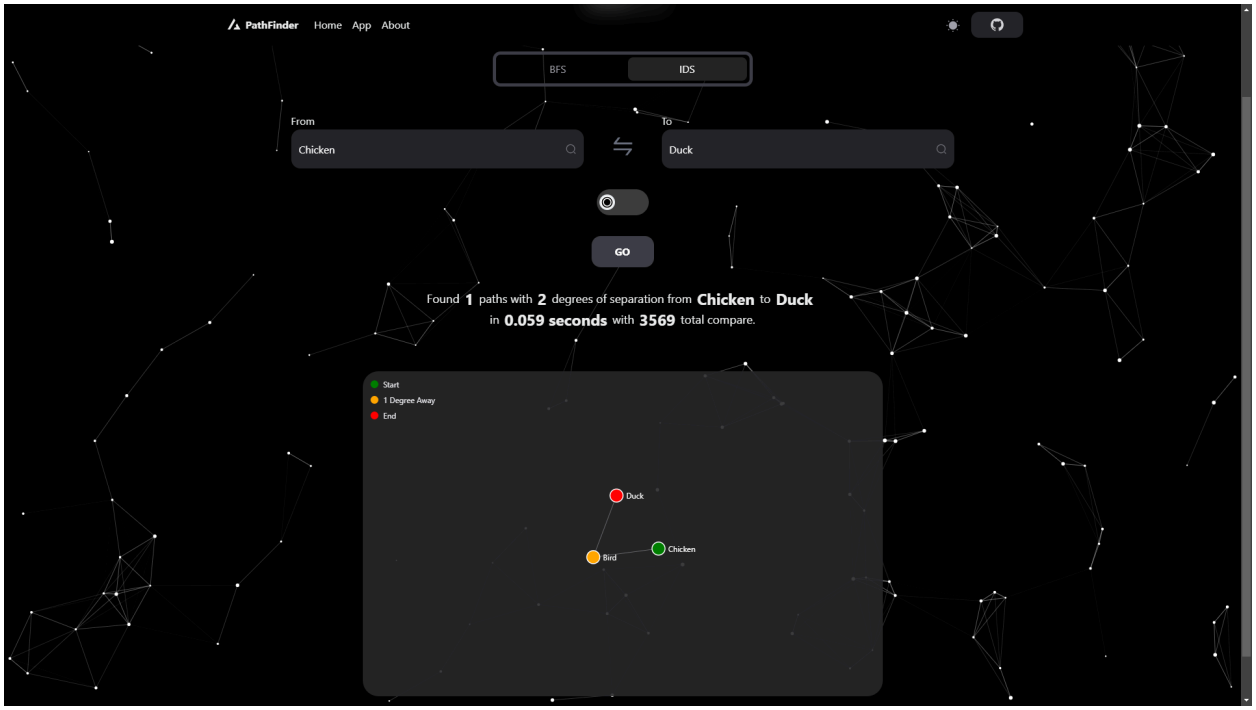
2. Hasil pengujian IDS:
    - IDS *single path* dari *"Jokowi"* menuju *"Prabowo Subianto"* (1 *degree*)



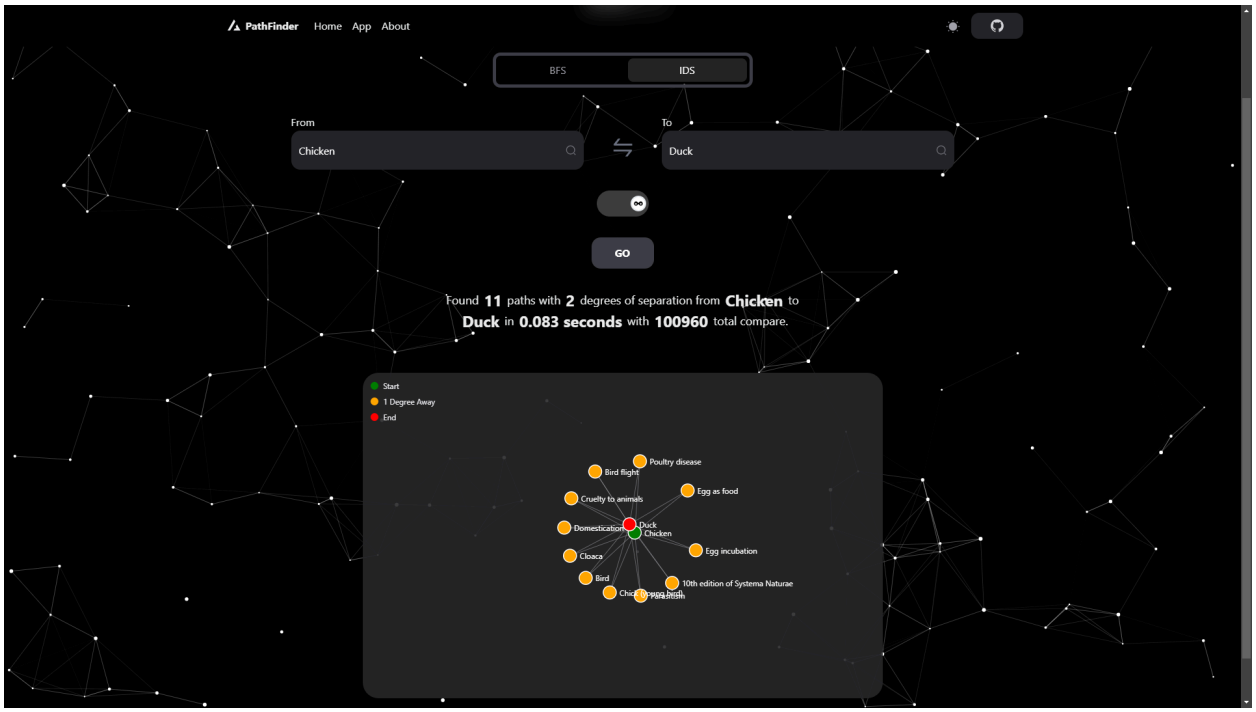
- IDS *multi path* dari *"Jokowi"* menuju *"Prabowo Subianto"* (1 *degree*)



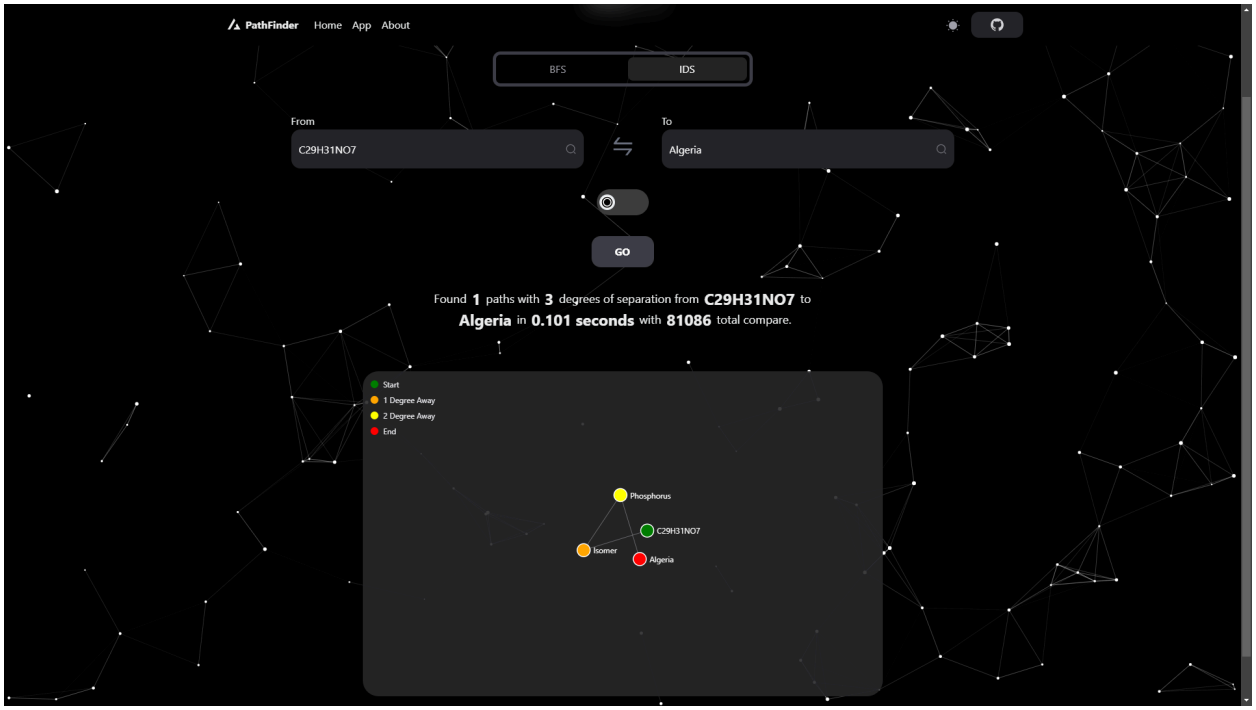
- IDS *single path* dari *"Chicken"* menuju *"Duck"* (2 *degree*)



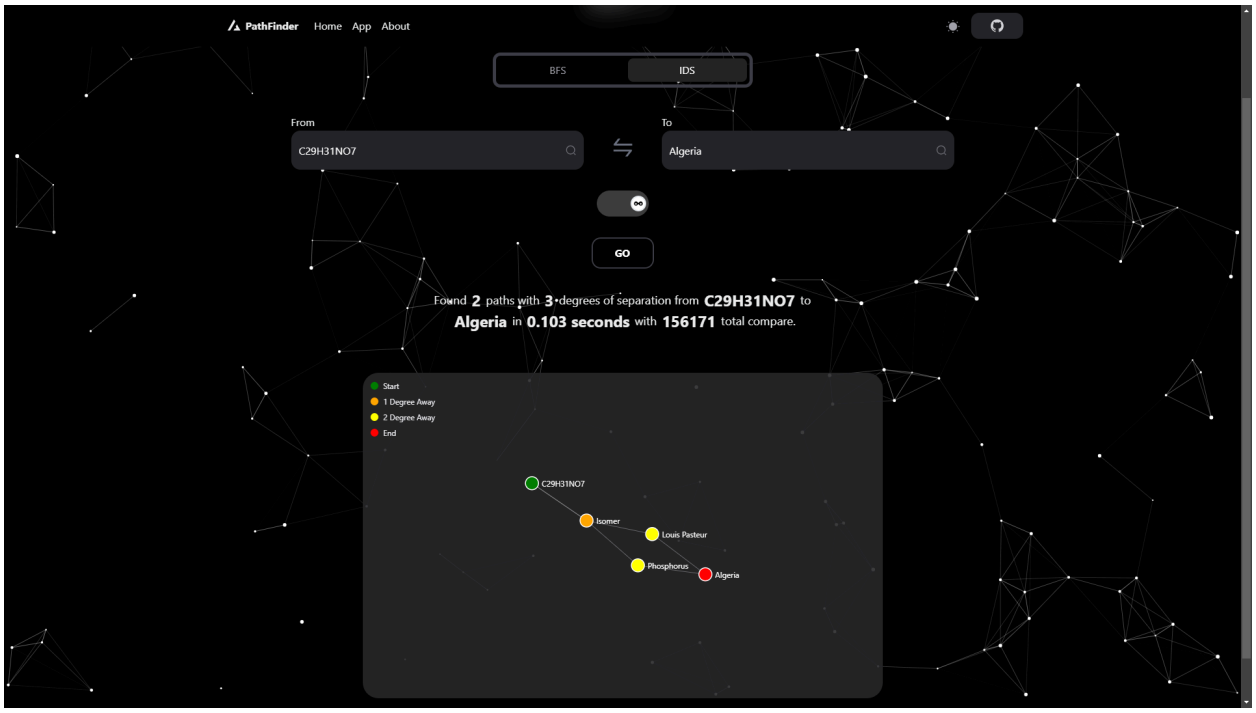
- IDS *multi path* dari *"Chicken"* menuju *"Duck"* (2 *degree*)



- IDS *single path* dari *"C29H31NO7"* menuju *"Algeria"* (3 *degree*)

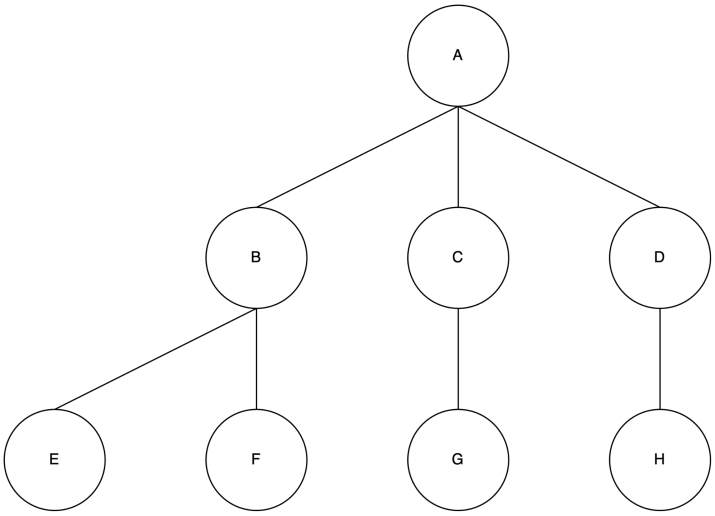


- IDS *multi path* dari *"C29H31NO7"* menuju *"Algeria"* (3 *degree*)



D. Analisis Hasil Pengujian

Berdasarkan hasil pengujian, dapat diketahui bahwa algoritma Breadth-First Search (BFS) dapat bekerja lebih cepat dibandingkan algoritma Iterative Deepening Search (IDS). Hal ini dikarenakan IDS membutuhkan pemeriksaan artikel yang lebih banyak dibandingkan BFS untuk pencarian artikel awal dan tujuan yang sama pada kedua algoritma. Contohnya :



Asumsikan A adalah artikel awal dan G artikel akhir. Untuk pencarian single path, algoritma BFS tidak selalu lebih cepat dari IDS karena ada kemungkinan bahwa artikel tujuan pada algoritma IDS ditemukan pada saat iterasi pertama dan pada BFS ditemukan pada iterasi akhir tetapi pada kedalaman yang sama. Pada single path, ketika ditemukan maka program akan langsung diberhentikan. Kalau pada gambar, dapat diasumsikan pada algoritma IDS, posisi G terletak pada E sedangkan pada algoritma BFS, posisi G terletak pada H. Sehingga ada faktor posisi url artikel pada saat scraping yang mempengaruhi untuk pencarian single path. Untuk pencarian multi path, jika artikel tujuan sudah ditemukan, maka program akan tetap berjalan untuk menelusuri hingga seluruh jalur dalam kedalaman tersebut selesai. Terjadi perbedaan jumlah artikel pada IDS dan BFS karena

Pada BFS

1. A (kedalaman 0)
2. B, C, D (kedalaman 1)
3. E, F, G, H (kedalaman 2) - G ditemukan pada kedalaman 2

Pada IDS

1. Iterasi 1: A (kedalaman 0)
2. Iterasi 2: A, B, C, D (kedalaman 1)
3. Iterasi 3: A, B, E, F, C, G, D, H (kedalaman 2) - G ditemukan pada kedalaman 2

IDS akan melakukan pemeriksaan artikel lebih banyak karena pada kedalaman 1 akan dihasilkan 3 dan pada kedalaman 2 akan dihasilkan 7. Sedangkan BFS lebih sedikit karena pada kedalaman 1 akan dihasilkan 3 dan pada kedalaman 2 akan dihasilkan 4. Total IDS lebih banyak daripada BFS karena IDS perlu melakukan backtracking dan adanya pembangunan node dari awal di setiap kedalamannya sedangkan BFS tidak perlu.

Algoritma BFS juga lebih cepat karena implementasi BFS secara asinkronus yang mendukung paralelisme dan berjalan secara konkuren sehingga prosesnya menjadi lebih cepat. Akan tetapi untuk single path, ada kondisi di mana jumlah pembacaan artikel lebih banyak dari IDS karena algoritma BFS yang memanfaatkan worker pada go sehingga pencarian yang dilakukan sangat cepat. Sedangkan pada IDS yang bersifat sinkronus, ketika artikel tujuan ditemukan, algoritma langsung berhenti dan program segera diterminasi. Meskipun algoritma BFS lebih cepat tetapi untuk pemakaian memori lebih boros dibandingkan IDS. Jadi akan ada kondisi dimana BFS akan lebih lambat dibandingkan IDS karena memory device yang terlalu penuh sehingga performa komputasi menurun.

BAB V
KESIMPULAN DAN SARAN

A. Kesimpulan

Dari tugas besar IF2211 Strategi Algoritma ini, telah diimplementasikan sebuah *website* untuk menentukan rute terpendek yang perlu ditempuh dari suatu hipertaut artikel Wikipedia (asal) menuju artikel Wikipedia (tujuan) dengan menggunakan algoritma BFS dan IDS. Website tersebut dibangun dengan menggunakan *framework NextJs* untuk *frontend*, dan bahasa pemrograman *Go* sebagai *backend* dengan spesifikasi fungsional dan tata cara penggunaan yang telah diuraikan pada bagian sebelumnya. *Website* ini juga menggunakan *database PostgreSQL* untuk mempercepat pencarian.

Dari analisis komprehensif yang disajikan dalam BAB IV, implementasi algoritma BFS dan IDS untuk menentukan lintasan terpendek antar artikel Wikipedia menunjukkan perbedaan kinerja antar algoritma yang jelas. Dapat disimpulkan bahwa algoritma BFS lebih unggul dari IDS dalam hal kecepatan eksekusi. Efisiensi ini terutama dikaitkan dengan jumlah pencarian artikel yang lebih sedikit yang dibutuhkan oleh BFS dibandingkan dengan IDS selama proses pencarian. Hal ini terjadi karena algoritma IDS menggunakan backtrack dan membangun graf dari awal sehingga lebih banyak memakan waktu sedangkan BFS hanya melanjutkan dari node terakhir. Penjelasan lebih detail dapat diamati pada bagian Analisis Hasil Pengujian.

Selain itu, ada juga pemanfaatan *database* yang memudahkan algoritma untuk tidak perlu melakukan *scraping* berulang untuk artikel yang sama sehingga proses pencarian dapat berlangsung lebih cepat. Pemakaian *Goroutine* pada algoritma BFS juga mempercepat jalannya algoritma karena mendistribusikan pekerjaannya menjadi beberapa thread.

B. Saran

Dalam mengembangkan tugas besar ini, ada beberapa strategi yang dapat diimplementasikan untuk meningkatkan efektivitas dalam menemukan jalur terpendek yang menghubungkan dua halaman Wikipedia. Pertama, adalah meningkatkan jumlah data yang tersedia dalam database dengan intensifikasi kegiatan scraping. Memperbanyak data akan memberikan peningkatan performa karena meminimalisir jumlah scraping ke *wikipedia* dengan langsung *load* dari database.

Kedua, sangat penting untuk mengoptimalkan penggunaan memori selama eksekusi program. Ini dapat dilakukan dengan mengimplementasikan teknik pemrograman yang lebih efisien atau memanfaatkan struktur data yang meminimalkan overhead memori. Dengan meminimalkan penggunaan memori, program tidak hanya akan berjalan lebih cepat tetapi juga lebih stabil, terutama saat diproses pada skala yang besar.

Selanjutnya, untuk meningkatkan kecepatan algoritma Breadth-First Search (BFS), dapat mengadopsi pendekatan double-ended BFS. Metode ini melibatkan pengaktifan simultan BFS dari kedua artikel awal dan tujuan. Dengan melakukan ini, setiap node atau artikel yang terhubung dengan artikel tujuan akan dianggap relevan dan berpotensi mempercepat proses pencarian dengan menciptakan titik pertemuan antara dua pencarian dari arah yang berlawanan. Teknik ini efektif karena memungkinkan algoritma untuk bertemu di tengah-tengah, sehingga mengurangi jumlah langkah yang diperlukan untuk mencapai tujuan.

LAMPIRAN

Pranala repository

https://github.com/mybajwk/Tubes2_PathFinder.git

Pranala video *youtube*

https://youtu.be/ppclPQN7KVw?si=Qd4_kx2yRoF9HUK-

Pranala database

https://drive.google.com/drive/folders/1LdG0G6ACDiKLjNpuCZNurzo-erRHypNv?usp=sharing

DAFTAR PUSTAKA

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

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