

**LAPORAN TUGAS BESAR II**  
**IF2211 STRATEGI ALGORITMA**

**PEMANFAATAN ALGORITMA BFS DAN IDS**  
**DALAM PERMAINAN WIKIRACE**



Kelompok JuBender

Anggota

13522055 Benardo

13522073 Juan Alfred Wijaya

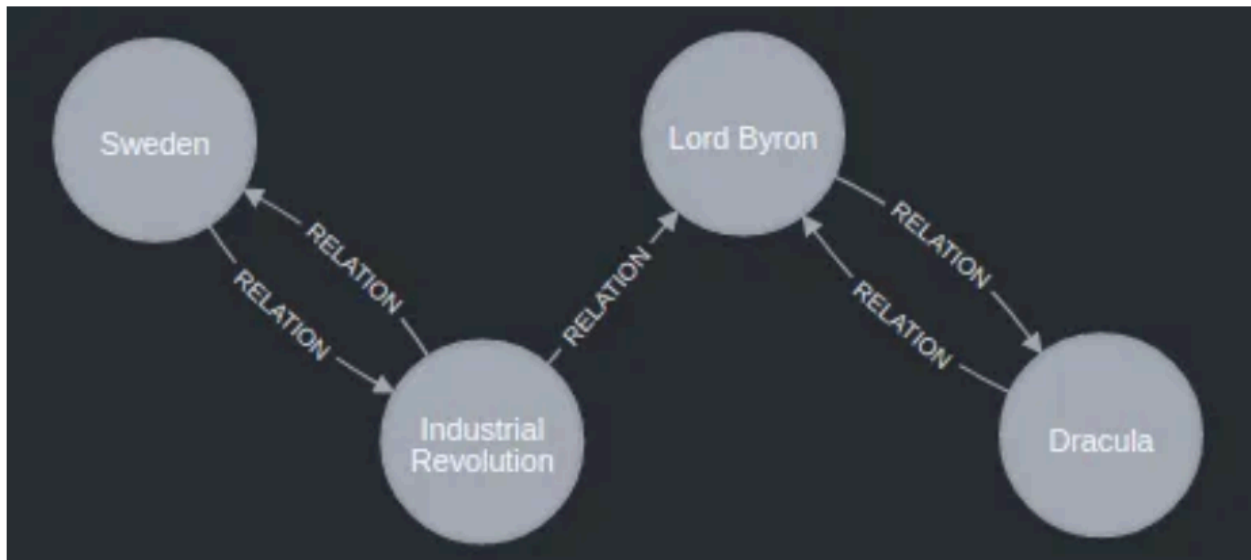
13522115 Derwin Rustanly

**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 kaskas pembuatan aplikasi berbasis website yang akan digunakan meliputi *React* yang merupakan *framework* dari bahasa pemrograman Javascript.

## BAB II

### LANDASAN TEORI

#### A. Penjelajahan Graf

Secara matematis, suatu graf dapat didefinisikan sebagai suatu representasi dari kumpulan objek diskrit dan relasi yang terdapat di antara objek-objek tersebut. Suatu graf memiliki 2 komponen utama, yakni simpul (vertex) dan sisi (edges). Suatu graf didefinisikan sebagai suatu tuple berelemen 2 dengan persamaan matematis  $G = (V, E)$  dengan  $V$  yang merupakan himpunan simpul yang terdapat pada graf dan  $E$  yang merupakan himpunan sisi yang menghubungkan sepasang simpul, atau dapat dituliskan sebagai

$$V = \{v_1, v_2, \dots, v_n\}$$

$$E = \{e_1, e_2, \dots, e_n\}$$

Penjelajahan (traversal) graf merupakan sebuah algoritma yang mengunjungi simpul-simpul di dalam graf dengan cara yang sistematis. Sebagaimana graf didefinisikan sebagai representasi suatu instansiasi persoalan, penjelajahan graf dapat diartikan sebagai proses pencarian solusi persoalan yang direpresentasikan dengan graf. Secara garis besar, terdapat 2 metode yang paling umum digunakan untuk melakukan penjelajahan graf, yakni metode pencarian melebar (*BFS; Breadth-First Search*) dan metode pencarian mendalam (*DFS; Depth-First Search*).

Dalam konteks pencarian solusi berbasis graf, algoritma pencarian solusi dapat dikategorikan menjadi dua jenis utama, yakni pencarian tanpa informasi (*uninformed* atau *blind search*) dan pencarian dengan informasi (*informed search*). Pencarian tanpa informasi, yang mencakup algoritma pencarian seperti DFS, BFS, DLS (*Depth Limited Search*), IDS (*Iterative Deepening Search*), dan *Uniform Cost Search* merupakan metode pencarian yang tidak mengandalkan data eksternal selain dari struktur graf tersebut sendiri. Di sisi lain, pencarian dengan informasi menggunakan pendekatan berbasis heuristik untuk memandu proses pencarian, sehingga memungkinkan pendekatan yang lebih terfokus dan efisien dengan mengidentifikasi simpul-simpul yang secara heuristik dinilai lebih mendekati solusi. Algoritma ini mencakup algoritma *Best First Search* dan  $A^*$ .

Selain itu, dalam proses pencarian solusi terdapat dua pendekatan terhadap representasi graf, yakni graf statis dan graf dinamis. Graf statis merupakan graf yang sudah terbentuk sebelum proses pencarian dilakukan, di mana graf direpresentasikan sebagai struktur data. Sementara itu, graf dinamis merupakan graf yang terbentuk saat proses pencarian dilakukan. Pada pendekatan ini, graf tidak tersedia sebelum pencarian melainkan graf dibangun selama pencarian solusi.

## **B. Algoritma BFS (*Breadth First Search*)**

*Breadth-First Search* (BFS) adalah algoritma penjelajahan atau pencarian graf yang memulai proses dari simpul akar dan mengeksplorasi semua tetangga dari simpul tersebut sebelum berpindah ke tetangga dari tetangga tersebut, dan seterusnya, berlapis-lapis. Algoritma ini mengadopsi strategi "lebar terlebih dahulu" dalam mengakses simpul-simpul graf. Secara teknis, BFS mengimplementasikan 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 sebagai dikunjungi. Selanjutnya, semua tetangga dari simpul ini yang belum dikunjungi dimasukkan ke dalam antrian, dan proses ini terus berulang. BFS sangat efektif untuk menemukan jalur terpendek dalam graf tidak berbobot karena menjamin bahwa ketika sebuah simpul pertama kali dikunjungi, jalur menuju simpul tersebut adalah yang terpendek.

Keunggulan BFS tidak hanya terbatas pada kemampuannya dalam mencari jalur terpendek, tetapi juga dalam aplikasinya pada berbagai masalah lain dalam ilmu komputer dan matematika. Contohnya termasuk menentukan konektivitas dalam graf, yaitu untuk menentukan apakah terdapat jalur yang menghubungkan dua simpul tertentu, atau untuk mengecek apakah graf tersebut terhubung sepenuhnya. 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 teknik pencarian yang menggabungkan efisiensi dari Breadth-First Search (BFS) dan kedalaman pencarian dari Depth-First Search (DFS) dalam satu framework yang koheren. IDS melakukan pencarian seperti DFS tetapi dengan batasan kedalaman yang meningkat secara bertahap. Pada dasarnya, IDS menjalankan DFS berulang kali dengan kedalaman yang dibatasi, yang dikenal sebagai Depth Limited Search (DLS), dan meningkatkan batas kedalaman tersebut setelah setiap iterasi lengkap melalui graf. Dengan melakukan ini, IDS dapat menggali lebih dalam ke dalam graf tanpa risiko terperangkap dalam kedalaman yang tak terbatas seperti yang mungkin terjadi pada DFS biasa.

Dalam Depth Limited Search, DFS dimodifikasi dengan menambahkan parameter kedalaman yang membatasi seberapa jauh pencarian bisa dilakukan. Ini mencegah rekursi tak terbatas ke dalam graf dan membantu mengelola konsumsi memori yang biasanya lebih besar dalam DFS standar. DLS sangat berguna dalam graf yang besar dan dalam, di mana DFS murni dapat terjebak dalam pencarian yang sangat lama tanpa menemukan solusi. DLS memastikan bahwa pencarian tetap berada dalam lingkup yang dapat dikelola dan mencegah penjelajahan yang tidak perlu pada tingkat kedalaman yang lebih rendah jika solusi tidak ada.

Iterative Deepening Search sangat efektif 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, 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 dan sering lebih disukai dibandingkan dengan pencarian lebar atau dalam murni.

Oleh karena itu, IDS menawarkan pendekatan yang seimbang dan adaptif untuk masalah pencarian kompleks, menggabungkan aspek terbaik dari kedua metode pencarian utama dalam ilmu komputer. IDS diakui karena keefektifannya dalam memastikan pencarian yang lengkap dan optimal sambil meminimalisir overhead memori, menjadikannya pilihan

yang sangat cocok untuk berbagai aplikasi praktis dalam pengembangan perangkat lunak dan sistem kecerdasan buatan.

#### **D. Aplikasi Website yang Dikembangkan**

Sebagaimana telah diuraikan pada bab 1, 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 untuk mengklik suatu hipertaut yang terdapat pada artikel Wikipedia asal menuju artikel tujuan. Proses pengumpulan data hipertaut yang terdapat pada artikel Wikipedia tersebut digunakan teknik *web scraping*.

*Web scraping* adalah proses ekstraksi data dari halaman web secara otomatis menggunakan bot atau program komputer. Teknik ini memungkinkan pengguna untuk mengumpulkan informasi yang terdapat dalam HTML dari berbagai situs web tanpa perlu melakukan pengumpulan data secara manual. Proses web scraping biasanya melibatkan akses ke halaman web dengan menggunakan HTTP atau protokol lainnya, dan kemudian menganalisis struktur HTML untuk mengekstrak data yang diinginkan.

Salah satu kakas yang sering digunakan untuk melakukan web scraping adalah GoColly. GoColly adalah sebuah library berbasis bahasa pemrograman Go yang dikembangkan untuk memudahkan proses scraping. GoColly menyediakan berbagai fitur yang berguna, termasuk kemampuan untuk mengikuti tautan (*link following*), menangani form, mengabaikan atau memfilter elemen tertentu, serta manajemen antrian untuk pengaturan pengunjungan halaman. 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 kakas *React* yang merupakan salah satu *framework* bahasa pemrograman JavaScript yang populer untuk membangun antarmuka pengguna yang responsif dan interaktif. React menyediakan komponen-komponen yang dapat digunakan kembali dan memiliki sistem manajemen *state* yang efisien, sehingga memudahkan pengembangan aplikasi web yang kompleks. Sementara itu, untuk bagian *Backend*,

digunakan bahasa pemrograman Go dan kakas GoColly untuk memproses data hipertaut dari artikel Wikipedia dengan algoritma BFS dan IDS untuk menentukan rute terpendek dari kedua artikel yang disediakan.

## **BAB III**

### **ANALISIS DAN PEMECAHAN MASALAH**

#### **A. Langkah - Langkah Pemecahan Masalah**

##### **1. Pendefinisian Masalah**

Aplikasi web yang akan kami kembangkan mempunyai tujuan untuk memungkinkan pengguna untuk mencari jalur terpendek yang menghubungkan dua halaman Wikipedia dengan menerima masukan halaman sumber dan tujuan dari pengguna. Pengguna dapat memilih antara menggunakan algoritma Breadth-First-Search (BFS) atau Iterative Deepening Search (IDS) untuk melakukan pencarian. Tujuan utama dari aplikasi ini adalah untuk memberikan solusi terpendek yang tepat dengan cepat dan akurat dengan juga menampilkan jalur yang ditemukan dalam sebuah antarmuka yang ramah pengguna.

##### **2. Menetapkan Desain Solusi**

Setelah melakukan pendefinisian masalah, kami kemudian menetapkan framework apa yang akan kami gunakan untuk frontend maupun backend. Untuk framework frontend akan digunakan framework React JS dan menggunakan styling Tailwind CSS. Untuk framework backend akan digunakan framework Gin dengan bahasa Go. Lalu, kami membuat desain antarmuka UI/ UX untuk web page kami. Dalam design web page, akan ada suatu form yang akan meminta input pengguna untuk judul halaman wikipedia yang menjadi titik awal dan titik akhir, dan juga algoritma apa yang akan digunakan. Lalu, saat melakukan pengisian input, akan ditampilkan rekomendasi melalui dropdown yang mirip dengan inputan user. Selain itu, ada juga tombol Search untuk mencari jalur terpendek. Hasil path terpendek akan ditampilkan dengan menggunakan library javascript d3.js.

##### **3. Integrasi dengan Backend**

Saat user menekan tombol search akan dilakukan Fetch API yang digunakan untuk mengirimkan permintaan dari front end ke backend. Data yang dikirim berupa judul halaman asal, judul halaman tujuan, dan algoritma yang dipilih. 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**

Selanjutnya, akan dilakukan pemrosesan data dengan menggunakan algoritma pencarian yang dipilih. Kemudian, akan dilakukan scraping terhadap page wikipedia, kemudian setelah hasil scraping diperoleh, hasil scraping akan disimpan dalam sebuah struktur data, lalu akan dilakukan pengecekan terhadap hasil scraping yang diperoleh, apakah ada yang sama dengan judul tujuan. Jika ditemukan ada yang sama, maka akan dikembalikan path untuk mencapai judul tersebut. Proses pencarian ini juga menerapkan pengoptimalan dengan menggunakan beberapa Goroutine (worker) untuk melakukan pencarian secara paralel dan konkuren. Dengan menerapkan pengoptimalan dan Goroutine ini diharapkan dapat menemukan solusi dengan lebih cepat.

#### **5. Caching**

Setelah melakukan web scraping untuk mendapatkan link-link yang terdapat dalam sebuah page wikipedia, untuk setiap page wikipedia akan disimpan isi link dari page. Penyimpanan caching ini dapat beragam, dapat berupa json dan juga database. Dengan adanya caching ini, akan mempersingkat waktu pencarian kedepannya, karena jika kedepannya ada melewati sebuah page wikipedia yang sudah terdapat di caching, maka tidak perlu melakukan web scraping lagi karena untuk melakukan scraping link akan menyebabkan pencarian menjadi lama dan juga banyak request yang ditolak oleh wikipedia.

#### **6. Respon ke Frontend**

Hasil pencarian jalur terpendek dan waktu yang dibutuhkan untuk melakukan pencarian lalu akan dikirim ke frontend dalam bentuk response. Di frontend kemudian akan menampilkan hasil ini kedalam bentuk teks maupun grafik. Untuk menampilkan grafik, kami menggunakan library javascript d3.js. Grafik yang ditampilkan juga sangat ramah pengguna, dimana grafik dapat digerakkan setiap simpulnya, dan juga dapat diperbesar dan diperkecil.

## 7. Pengujian

Setelah program selesai dirancang , lalu kami melakukan pengujian terhadap program kami. Pengujian dilakukan dengan mencoba beberapa testcase, lalu mengecek hasilnya dengan referensi web wiki racer sixdegrees wikipedia. Selain itu , juga dilakukan pengujian dan revisi beberapa kali untuk mendapatkan hasil dengan waktu yang cepat , sehingga memenuhi bonus untuk mendapatkan hasil di bawah 1 menit.

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

### 1. Pemetaan menjadi Elemen Algoritma BFS

Elemen:

- Node Awal : Artikel Wikipedia awal.
- Node Tujuan : Artikel Wikipedia tujuan.
- Tujuan : Menemukan jalur terpendek dari artikel awal ke artikel tujuan dengan melalui link yang ada pada setiap artikel.
- Queue : Menyimpan artikel yang harus dikunjungi dan diperiksa.
- Visited Set : Mencatat semua artikel yang sudah dikunjungi untuk menghindari siklus dan pengulangan.
- Children : Setiap artikel yang ditautkan dari artikel yang sedang dikunjungi.

Proses:

1. Mulai dari artikel awal, masukkan ke dalam queue.
2. Keluarkan artikel terdepan dari queue.
3. Ambil semua artikel yang ditautkan (yang belum dikunjungi) dan cek apakah artikel yang ditautkan merupakan artikel tujuan. Jika ya, maka keluar dari proses. Jika bukan, tandai artikel yang ditautkan sebagai dikunjungi dan masukkan ke dalam queue.
4. Ulangi proses dari nomor 2 sampai menemukan artikel tujuan atau queue kosong.

### 2. Pemetaan menjadi Elemen Algoritma IDS

Elemen:

- Node Awal : Artikel Wikipedia awal.

- Node Tujuan : Artikel Wikipedia tujuan.
- Tujuan : Menemukan jalur terpendek dari artikel awal ke artikel tujuan dengan melalui link yang ada pada setiap artikel.
- Depth Limit: Batasan kedalaman yang bertambah secara iteratif.
- Stack: Menyimpan node untuk proses depth limited search dalam setiap iterasi kedalaman.
- Visited Set: Mencatat semua artikel yang sudah dikunjungi untuk menghindari siklus dan pengulangan.

Proses:

1. Batasan kedalaman dimulai dari 0.
2. Lakukan DLS (depth limited search dengan kedalaman 0).

Proses DLS:

- Masukkan artikel awal ke stack.
  - Ambil artikel paling atas dari stack.
  - Cek apakah artikel yang diambil sudah berada di kedalaman maksimum (mencapai limit). Jika sudah, ambil artikel paling atas selanjutnya, jika belum, maka proses lanjut.
  - Ambil semua artikel yang ditautkan (yang belum dikunjungi) dan cek apakah artikel yang ditautkan merupakan artikel tujuan. Jika ya, maka keluar dari proses. Jika bukan, tandai artikel yang ditautkan sebagai dikunjungi dan masukkan ke dalam stack.
  - Ulangi proses dari urutan kedua hingga stack kosong.
3. Jika masih belum menemukan artikel tujuan maka lakukan kembali DLS dengan kedalaman maksimum (limit) ditambah satu dari sebelumnya.

## **C. Fitur Fungsional dan Arsitektur Aplikasi**

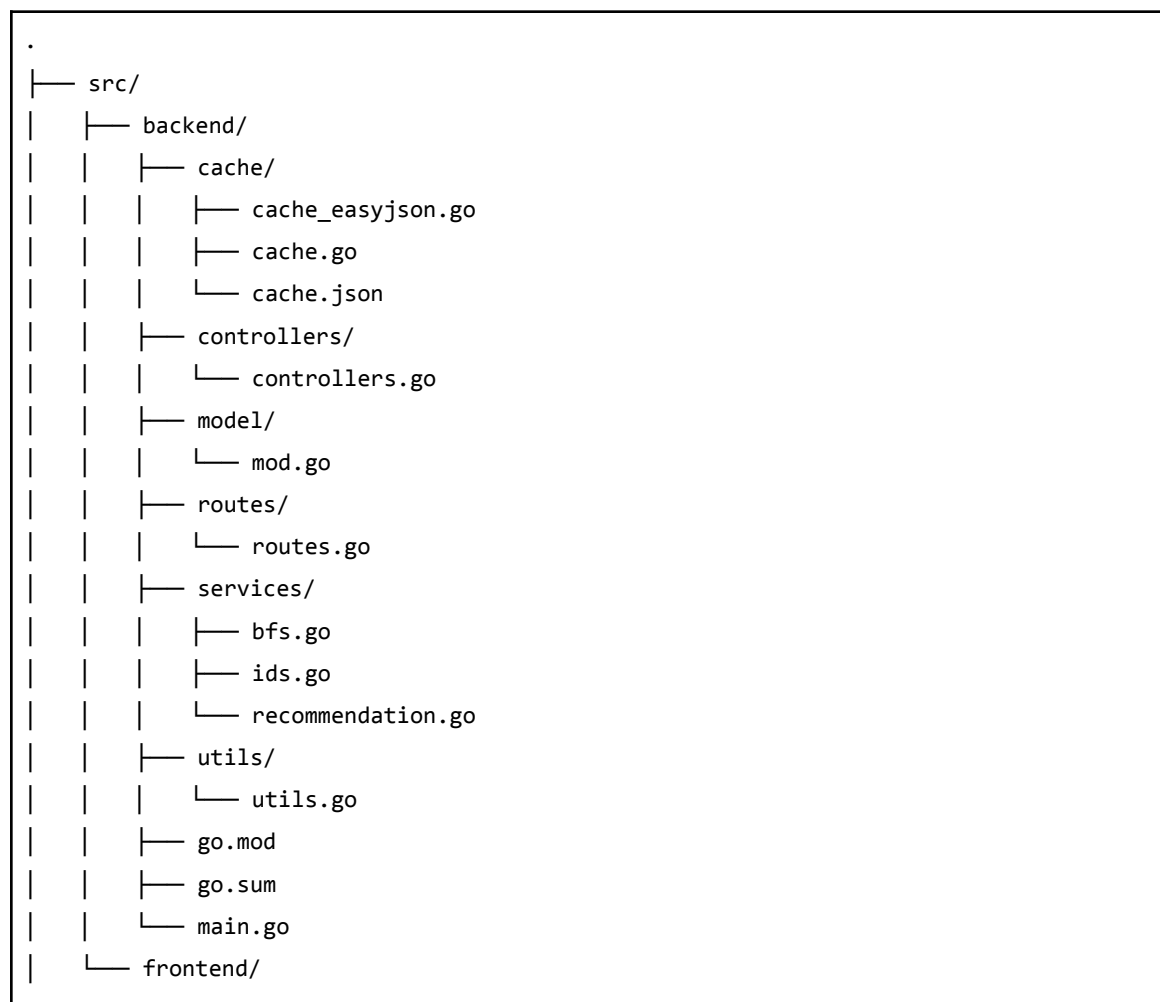
### **1. Fitur Fungsional**

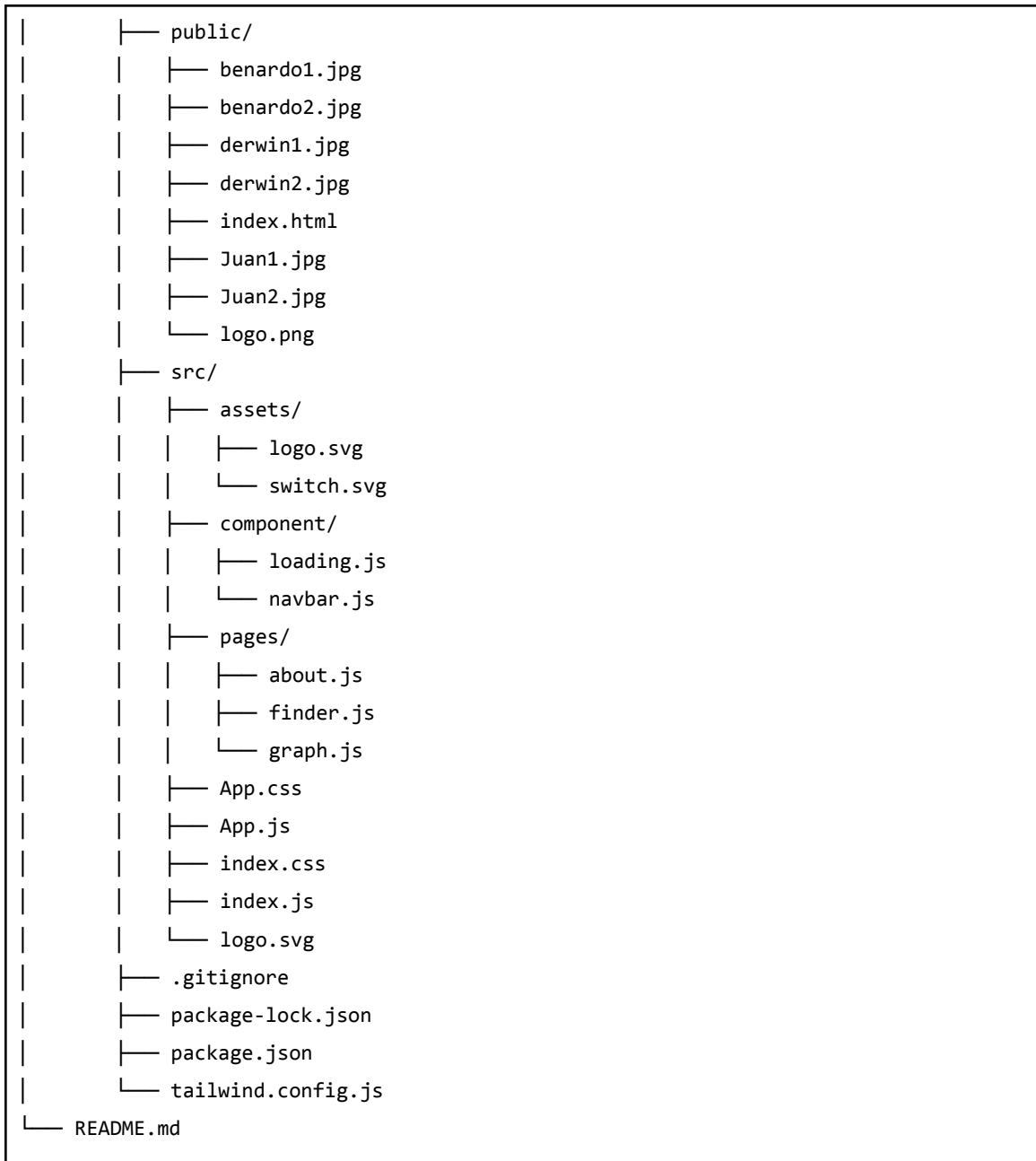
Ada beberapa fitur fungsional yang terdapat pada aplikasi website ini, antara lain:

- Tombol switch untuk memilih antara algoritma BFS atau IDS

- Dropdown rekomendasi pada field input source dan destination untuk memudahkan user untuk memilih judul page yang ada di wikipedia. Dropdown rekomendasi ini akan berubah saat pengguna melakukan pengetikan input field.
- Tombol switch untuk source dan destination , sehingga bisa melakukan pertukaran secara langsung antara source dan destination.
- Tombol Search 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 , dan juga dapat di zoom in dan zoom out.

## 2. Arsitektur Aplikasi





Untuk aplikasi ini, kami menggunakan React Js + Tailwind CSS untuk frontend, kemudian untuk backend, 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.

Dalam folder backend memiliki struktur yang modular dengan pembagian yang jelas antara controllers, services, routes, models dan lain-lain. Rincian folder backend dapat dilihat di bawah ini:

- `cache/`: Direktori ini mengandung implementasi cache (terlihat dari file seperti `cache.go` dan `cache_easyjson.go`), serta file `cache.json` yang mungkin berisi data yang disimpan dalam cache untuk mempercepat proses pengambilan data.
- `controllers/`: Berisi file `controllers.go`, yang biasanya mengatur logika kontroler untuk mengelola request yang masuk dan mengirim response kembali ke klien.
- `model/`: Direktori `model/` dengan file `mod.go` diduga mendefinisikan struktur data yang digunakan dalam aplikasi ini.
- `routes/`: File `routes.go` dalam folder ini bertanggung jawab untuk mendefinisikan rute HTTP dan pemetaannya ke fungsi kontroler yang relevan.
- `services/`: Berisi kumpulan file seperti `bfs.go`, `ids.go`, dan `recommendation.go` yang mungkin masing-masing menangani pencarian berdasarkan algoritma Breadth-First Search, Iterative Deepening Search, dan fungsi untuk memberikan rekomendasi.
- `utils/`: Direktori `utils/` menyimpan utilitas bantu dalam `utils.go` yang digunakan di seluruh aplikasi backend untuk fungsi umum atau berulang.
- `main.go`: Ini adalah entry point aplikasi backend, dimana server diinisialisasi dan konfigurasi aplikasi diatur.

Dalam folder frontend memiliki struktur folder sebagai berikut:

- `public/`: Berisi aset statis seperti gambar dan file `index.html`, yang merupakan template HTML dasar untuk aplikasi frontend.
- `src/assets/`: Mengandung aset seperti `logo.svg` dan `switch.svg` yang akan digunakan dalam UI aplikasi.
- `src/component/`: Direktori ini menyimpan komponen-komponen React yang dibuat khusus seperti `loading.js` dan `navbar.js` untuk bagian yang dapat digunakan kembali di berbagai halaman aplikasi. Hal ini berfungsi agar aplikasi ini menerapkan prinsip modular.

- `src/pages/`: Mengandung file seperti `about.js`, `finder.js`, dan `graph.js` yang mendefinisikan berbagai halaman atau 'views' dari aplikasi web ini.
- `src/App.css`, `src/App.js`, dan `src/index.js`: Ini adalah file utama yang mengatur styling global, komponen App utama, dan entry point untuk React di dalam aplikasi frontend.
- `tailwind.config.js`: File konfigurasi untuk Tailwind CSS, sebuah framework styling yang digunakan untuk mendesain tampilan aplikasi dengan efisien.

#### D. Contoh Ilustrasi Kasus

Salah satu contoh kasus yang dapat diselesaikan dengan program berbasis website yang mengimplementasikan algoritma BFS dan IDS adalah menentukan jumlah hipertaut yang perlu diklik dari artikel Wikipedia dengan judul Indonesia ([en.wikipedia.org/wiki/Indonesia](https://en.wikipedia.org/wiki/Indonesia)) menuju artikel berjudul Jakarta ([en.wikipedia.org/wiki/Jakarta](https://en.wikipedia.org/wiki/Jakarta)).

Dengan algoritma BFS, artikel asal akan disimpan sebagai simpul akar yang ditambahkan pada antrian (*queue*), kemudian struktur data visited set yang merupakan map akan menginisialisasikan simpul akar sebagai true, sehingga simpul tersebut tidak akan dikunjungi pada penjelajahan graf berikutnya. Kemudian, akan dibangkitkan seluruh simpul anak yang merupakan seluruh hipertaut artikel yang dapat dituju dari artikel asal yang diimplementasikan dengan menambahkan keseluruhan artikel tersebut ke dalam antrian. Selama antrian tidak kosong dan tidak ditemukan artikel tujuan, akan dilakukan iterasi terhadap setiap simpul anak yang mewakili hipertaut artikel berdasarkan urutan kebangkitan simpul tersebut kemudian visited set akan menandai simpul tersebut sebagai true, sehingga simpul tersebut tidak akan dikunjungi pada penjelajahan berikutnya.

Dengan algoritma IDS, artikel asal akan disimpan pula sebagai simpul akar yang ditambahkan pada tumpukan (*stack*), sama halnya dengan BFS, struktur data visited set akan menginisialisasi simpul akar sebagai true, sehingga simpul tersebut tidak akan dikunjungi pada penjelajahan graf berikutnya. Kemudian, akan dilakukan IDS yang merupakan DLS dengan kedalaman yang meningkat apabila artikel tujuan belum ditemukan. Adapun, metode penjelajahan simpul menggunakan algoritma pop pada *stack* yakni mengambil artikel yang merupakan elemen teratas *stack*, kemudian menambahkan simpul ekspansi yang merupakan seluruh hipertaut artikel yang dapat dituju dari artikel yang bersangkutan. Selama tumpukan

belum kosong dan tidak ditemukan simpul tujuan, iterasi akan dilakukan secara “mendalam” ke dalam setiap simpul anak yang diekspan, kemudian visited set akan menandai simpul tersebut sebagai true, sehingga simpul tersebut tidak akan dikunjungi pada penjelajahan berikutnya, hingga ditemukan simpul tujuan atau dicapai batas kedalaman maksimum yang ditentukan. Apabila simpul tujuan belum ditemukan, akan dilakukan *backtracking* pada simpul sebelumnya yang secara tidak langsung telah diimplementasikan dengan algoritma pop yang telah diuraikan.



## BAB IV

### IMPLEMENTASI DAN PENGUJIAN

#### A. Spesifikasi Teknis Program

##### 1. Implementasi Struktur Data

###### a. Algoritma BFS

Pada algoritma BFS (*Breadth First Search*) diimplementasikan struktur data antrian (*queue*) yang berfungsi untuk mengatur urutan pembangkitan simpul dalam proses penjelajahan graf. Algoritma ini dimulai dengan menempatkan simpul awal ke dalam antrian. Setelah simpul awal dimasukkan, BFS memasuki loop utama yang terus berlangsung selama antrian belum kosong. Dalam setiap iterasi dari loop ini, simpul yang berada di depan antrian diambil dan diproses. Proses ini biasanya melibatkan pencatatan atau pemrosesan simpul tersebut, serta penambahan semua simpul tetangga yang belum dikunjungi ke dalam antrian. simpul tetangga ini akan dieksplorasi dalam iterasi berikutnya, memastikan bahwa penjelajahan terjadi secara melebar (*breadth-first*) sebelum mendalam.

Dalam hal penentuan rute yang melibatkan dua hipertaut artikel Wikipedia, antrian digunakan untuk secara efisien mengelola penjelajahan artikel secara sistematis dari satu ke artikel lain. Algoritma BFS dimulai dengan memasukkan artikel sumber ke dalam antrian dan menandainya sebagai dikunjungi. Kemudian, loop utama BFS dijalankan, di mana artikel di depan antrian diambil dan semua hipertaut yang terhubung diperiksa. Jika suatu hipertaut mengarah pada artikel tujuan, rute tersebut telah ditemukan. Jika tidak, setiap artikel yang terhubung dan belum ditandai sebagai dikunjungi ditambahkan ke dalam antrian untuk dieksplorasi lebih lanjut. Ini memastikan bahwa semua artikel yang mungkin yang terhubung secara langsung atau tidak langsung dengan artikel sumber akan diperiksa dalam urutan yang terorganisir. Dengan cara ini, BFS dapat secara efektif mencari melalui jaringan artikel yang luas, menemukan jalur terpendek atau jalur yang

mungkin antara dua halaman, berkat struktur antrian yang memprioritaskan penjelajahan berdasarkan urutan penambahan artikel.

b. Algoritma IDS

Pada algoritma IDS, digunakan struktur data tumpukan (*stack*) untuk mengelola simpul-simpul selama pencarian, yang serupa dengan DFS. Dalam IDS, DFS dijalankan berulang kali dengan batas kedalaman yang meningkat setiap iterasinya. Penjelajahan graf dimulai dari simpul awal, dan tumpukan digunakan untuk menyimpan simpul saat algoritma mencoba mencapai kedalaman yang lebih dalam pada setiap siklus. Setiap kali batas kedalaman tercapai tanpa menemukan solusi, pencarian diulang dari simpul awal dengan batas yang lebih tinggi. Dengan menggunakan tumpukan, IDS secara efektif menelusuri lebih dalam pada graf dengan mengulang dari awal dan memperluas batas kedalaman secara bertahap, memungkinkan pencarian menyeluruh secara bertahap dengan menggunakan ruang memori yang relatif sedikit. Ini memastikan bahwa pencarian tetap ringan dan efisien, bahkan dalam menghadapi ruang pencarian yang sangat besar atau kompleks, karena setiap kedalaman dijelajahi secara menyeluruh sebelum pindah ke tingkat berikutnya, menciptakan pendekatan yang sistematis dan terkontrol dalam pencarian solusi.

Dalam hal penentuan rute yang melibatkan dua hipertaut artikel Wikipedia, tumpukan digunakan dalam algoritma IDS untuk membantu mengelola proses penjelajahan secara efisien. Algoritma ini dimulai dengan mengatur kedalaman awal yang rendah dan menggunakan tumpukan untuk mengendalikan urutan penelusuran simpul dalam graf yang merepresentasikan koneksi antar artikel. Setiap artikel dimasukkan ke dalam tumpukan sebagai simpul awal, kemudian IDS mencoba menelusuri artikel dengan kedalaman yang terbatas. Ketika mencapai batas kedalaman tanpa menemukan artikel tujuan, IDS mengulang pencarian dari simpul awal dengan meningkatkan batas kedalaman. Dalam setiap iterasi, semua simpul yang dijelajahi pada kedalaman tertentu ditambahkan ke tumpukan, dan kemudian simpul-simpul

ini dieksplorasi satu per satu dari elemen teratas tumpukan (*top of stack*). Jika sebuah simpul memiliki tautan ke artikel lain, tautan tersebut ditambahkan ke tumpukan untuk dieksplorasi, jika masih dalam batas kedalaman yang diizinkan. Dengan demikian, IDS menggunakan tumpukan secara strategis dan bertahap menggali lebih dalam ke dalam graf artikel Wikipedia, sehingga memastikan bahwa setiap potensi rute diantara dua artikel ditelusuri secara menyeluruh.

## 2. Implementasi Fungsi dan Prosedur

Berikut merupakan implementasi fungsi dan prosedur dalam bentuk pseudocode.

```
FUNCTION unwrapParentMap(targetURL: string, parentMap: Map of string to
List of string) → List of List of string
{ Initialization }
unwrappedPath ← empty list of list of strings
seenPaths ← initialize empty map of string to boolean
{ Process each potential parent URL from the map }
FOR EACH unwrapURL IN parentMap[targetURL] DO
    tempPath ← list containing title-formatted version of targetURL
    without the prefix
    fullPath ← empty string
    url ← unwrapURL
    { Unwrap the parent map with a loop }
    WHILE url not empty DO
        trimmedLink ← remove prefix "https://en.wikipedia.org/wiki/" from
url
        formattedTitle ← format trimmedLink into a title
        prepend formattedTitle to tempPath
        append formattedTitle to fullPath
        IF url = parentMap[url][0] THEN
            BREAK
        url ← parentMap[url][0]
    { Check for unique path and add to results }
    IF fullPath not in seenPaths THEN
        add tempPath to unwrappedPath
        mark fullPath as seen in seenPaths
RETURN unwrappedPath
```

```
FUNCTION HandleBFS(startTitle: string, targetTitle: string, multiple:
boolean) → Map of string to any
```

```
{ Initialization }
parentMap ← new Map of string to List of string
totalLinksSearched ← 0
totalRequest ← 0
startURL ← "https://en.wikipedia.org/wiki/" +
utils.EncodeToPercent(startTitle)
targetURL ← "https://en.wikipedia.org/wiki/" +
utils.EncodeToPercent(targetTitle)
startTime ← time.Now()
{ Execute BFS to find the path }
bfs(startURL, targetURL, multiple, parentMap, cache.GlobalCache.Data,
totalLinksSearched, totalRequest)
elapsed ← time.Since(startTime)
path ← unwrapParentMap(targetURL, parentMap)
{ Return function value with search information }
RETURN Map with "from": utils.FormatToTitle(startTitle),
               "to": utils.FormatToTitle(targetTitle),
               "time_ms": elapsed.Milliseconds(),
               "total_link_searched": totalLinksSearched,
               "total_scrap_request": totalRequest,
               "path": path
```

```
FUNCTION bfs(startURL: string, targetURL: string, multiple: boolean,
parentMap: Map of string to List of string, cache: Map of string to List
of string, totalLinksSearched: integer reference, totalRequest: integer
reference)
```

```
{ Check initial condition if target matches start }
IF startURL = targetURL THEN
    parentMap[targetURL] ← [startURL]
    totalLinksSearched ← 1
    RETURN
{ Initialize data structures and variables for BFS }
visited ← new sync.Map
queue1 ← new channel of Article with capacity 7000000
queue2 ← new channel of Article with capacity 7000000
currentDepth ← 0
mutex ← new sync.Mutex
targetFound ← new atomic integer initialized to 0
```

```

depthFound ← new atomic integer initialized to -1
runningQueue ← reference to queue1
excludeRegex ← compile regex to exclude non-article pages
c ← newCollyCollector for domain "en.wikipedia.org"

DEFINE helper functions for enqueue, dequeue, addParent, findDepth using
parentMap and other local variables

{ Define event handler for HTML links }
c.OnHTML("a[href]", FUNCTION(e: colly.HTMLElement)
    link ← get absolute URL from e
    trimmedLink ← remove prefix "https://en.wikipedia.org" from link
    IF link starts with "/wiki/" AND NOT excludeRegex matches trimmedLink
    THEN
        linkEncoded ← encode link URL
        parentUrlEncoded ← encode parent URL from e
        IF NOT visited.LoadOrStore(link, true) THEN
            totalLinksSearched += 1
            depth ← findDepth(parentUrlEncoded) + 1
            article ← new Article with Url: link, Depth: depth
            { Handle queue and add parent based on depth }
            addParent(link, parentUrlEncoded)
            enqueue(article to runningQueue based on depth)
            IF link = targetURL THEN
                targetFound ← 1
                depthFound ← depth - 1
                PRINT ">> Found MINIMAL path at:", parentUrlEncoded
                PRINT "> Target:", link

{ Manage concurrency and process articles from queues }
FOR i FROM 1 TO 20 DO
    START new goroutine
    WHILE atomic targetFound = 0 OR (multiple AND currentDepth <=
depthFound) DO
        IF runningQueue is empty THEN
            SWITCH queues and increment currentDepth
        article, ok ← dequeue(runningQueue)
        IF ok THEN
            { Process link, visit using collector, and handle caching }
            IF cached THEN
                fetch from cache and process cached links

```

```

ELSE
    c.Visit(article.Url)
{ Wait for all goroutines to finish and close queues }
WAIT for all goroutines
CLOSE queue1
CLOSE queue2

```

---

```

FUNCTION HandleIDS(startTitle: string, targetTitle: string, multiple:
boolean) → Map of string to any
{ Initialization }
startURL ← "https://en.wikipedia.org/wiki/" +
utils.EncodeToPercent(startTitle)
targetURL ← "https://en.wikipedia.org/wiki/" +
utils.EncodeToPercent(targetTitle)
parentMap ← new Map of string to List of string
totalLinksSearched ← 0
totalRequest ← 0
startTime ← time.Now()

{ Execute IDS }
ids(startURL, targetURL, multiple, parentMap, totalLinksSearched,
totalRequest, cache.GlobalCache.Data)

elapsed ← time.Since(startTime)
{ Return function value with search information }
RETURN Map with "from": utils.FormatToTitle(startTitle),
               "to": utils.FormatToTitle(targetTitle),
               "time_ms": elapsed.Milliseconds(),
               "total_link_searched": totalLinksSearched,
               "total_scrap_request": totalRequest,
               "path": unwrapParentMap(targetURL, parentMap)

```

---

```

FUNCTION ids(startURL: string, targetURL: string, parentMap: Map of string
to string, totalLinksSearched: integer reference, totalRequest: integer
reference, cache: Map of string to List of string)
excludeRegex ←
regexp.MustCompile(`^/wiki/(File:|Category:|Special:|Portal:|Help:|Wikiped
ia:|Talk:|User:|Template:|Template_talk:|Main_Page)`)
checkMap ← new Map of string to boolean
targetFound ← false
i ← 0
WHILE NOT targetFound DO

```

```

    targetFound ← dls(startURL, targetURL, i, checkMap, parentMap,
totalLinksSearched, totalRequest, cache, excludeRegex)
    PRINT "Done iterate:", i
    i++

```

```

FUNCTION dls(startURL: string, targetURL: string, limit: integer,
multiple: boolean, checkMap: Map of string to boolean, parentMap: Map of
string to List of string, totalLinksSearched: integer reference,
totalRequest: integer reference, cache: Map of string to List of string,
excludeRegex: regexp.Regexp) → boolean

```

```

IF startURL = targetURL THEN

```

```

    parentMap[targetURL] ← [startURL]

```

```

    totalLinksSearched ← 1

```

```

    RETURN true

```

```

visited ← new Map of string to boolean

```

```

stack ← List of Article initialized with {Url: startURL, Depth: 0}

```

```

parentMap[startURL] ← [""]

```

```

found ← false

```

```

WHILE stack not empty DO

```

```

    nextArticle ← stack last element

```

```

    nextURL ← utils.WikipediaUrlEncode(nextArticle.Url)

```

```

    REMOVE last element from stack

```

```

    IF visited contains nextURL THEN

```

```

        CONTINUE

```

```

    IF nextURL = targetURL THEN

```

```

        PRINT "Found:", nextURL

```

```

        found ← true

```

```

        IF NOT multiple THEN

```

```

            RETURN true

```

```

    IF NOT checkMap contains nextURL THEN

```

```

        totalLinksSearched += 1

```

```

    checkMap[nextURL] ← true

```

```

    IF nextArticle.Depth = limit THEN

```

```

        CONTINUE

        scrapResult ← scrapArticles(nextURL, cache, totalRequest,
excludeRegex)
        visited[nextURL] ← true
        stack ← append stack with wrapToArticle(nextArticle, scrapResult,
parentMap)

RETURN found

```

## B. Penjelasan Tata Cara Penggunaan Program

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

```
git clone https://github.com/DerwinRustanly/Tubes2_JuBender.git
```

2. Pengguna dapat mengunduh cache yang disediakan pada tautan google drive berikut untuk mempercepat hasil pencarian, kemudian pengguna menyimpan file cache.json pada folder src/backend/cache

<https://bit.ly/cacheJuBender>

3. Jalankan perintah berikut untuk melakukan build website

```
docker-compose up --build
```

4. Jalankan browser dan buka laman untuk menjalankan website

```
localhost:3000
```



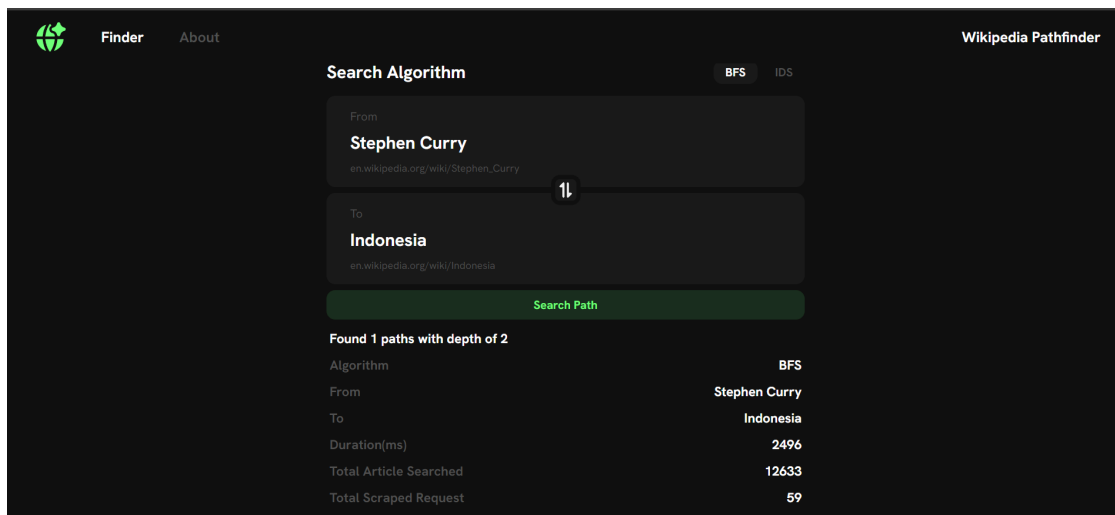
5. Setelah pengguna berhasil menjalankan website, pengguna dapat memilih algoritma pencarian, baik dengan menggunakan algoritma BFS maupun IDS.
6. Setelah pengguna memilih algoritma pencarian, pengguna menuliskan judul artikel Wikipedia asal dan tujuan, program juga akan memberikan rekomendasi artikel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

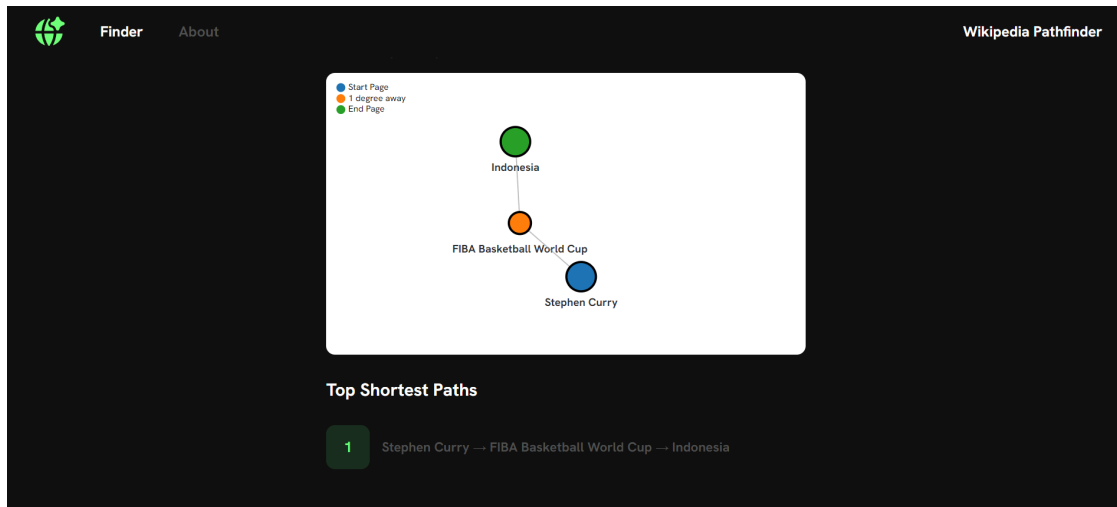
### C. Hasil Pengujian

1. TestCase 1 (degree 2)

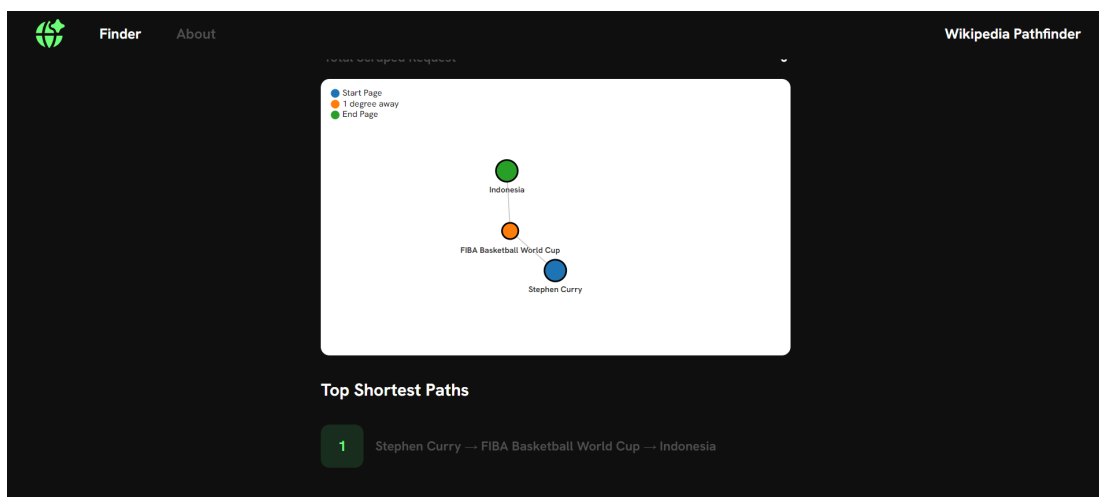
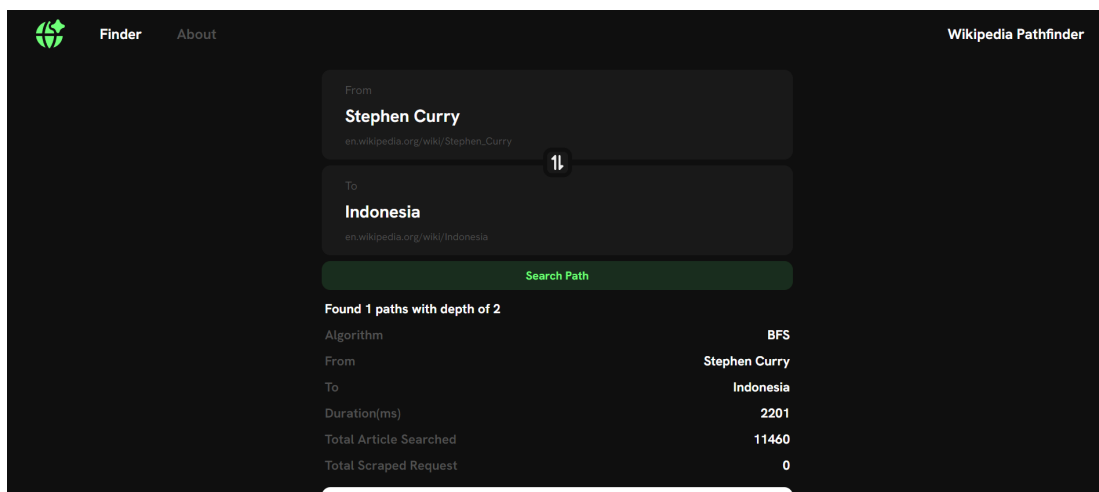
#### BFS Single path

Sebelum caching



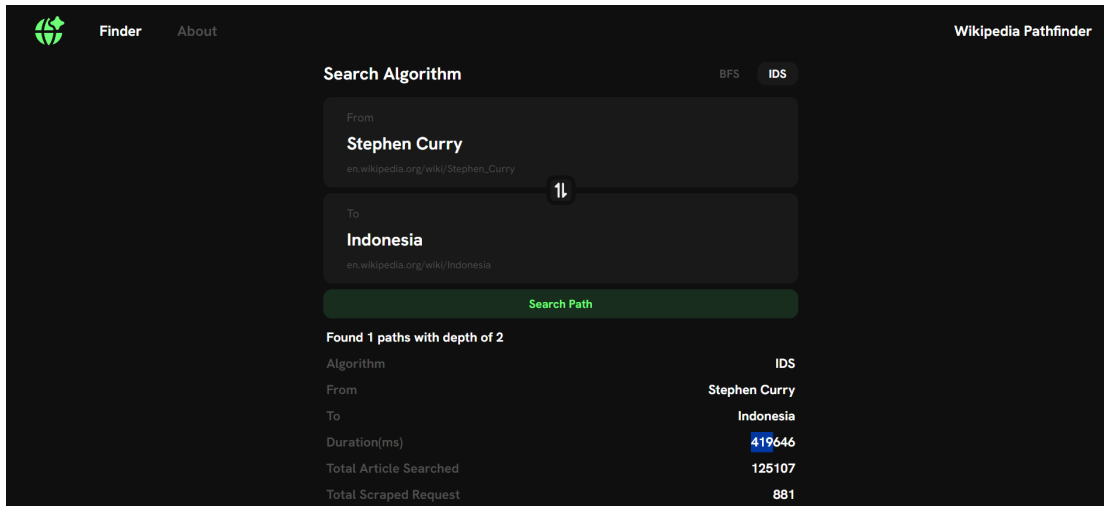


Sesudah caching



## IDS Single path

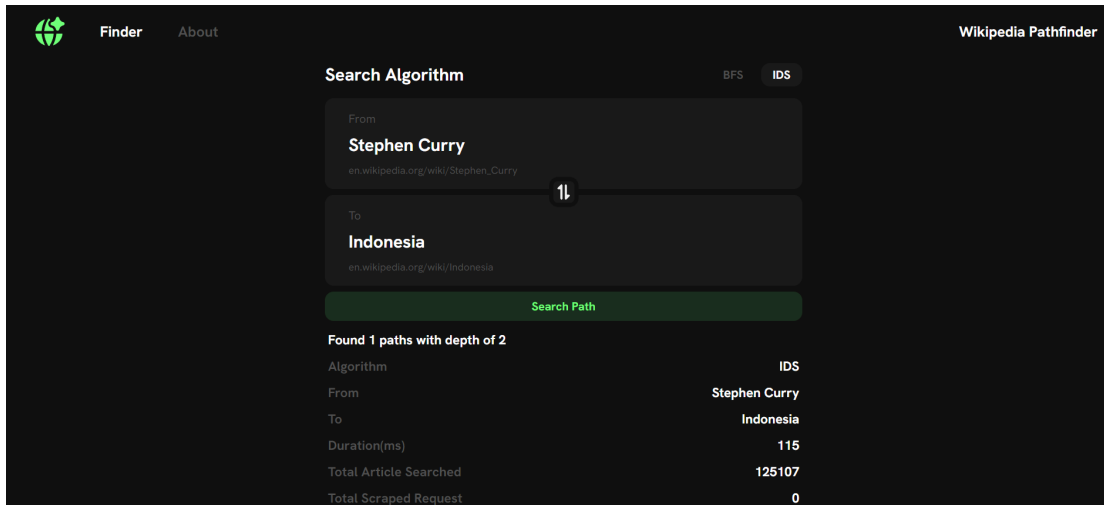
Sebelum caching



The screenshot shows the Wikipedia Pathfinder interface. The search algorithm is set to IDS. The search path is from Stephen Curry to Indonesia. The results show 1 path with a depth of 2. The duration is 419646 ms, and the total article searched is 125107. The total scraped request is 881.

Found 1 paths with depth of 2	
Algorithm	IDS
From	Stephen Curry
To	Indonesia
Duration(ms)	419646
Total Article Searched	125107
Total Scraped Request	881

Sesudah caching

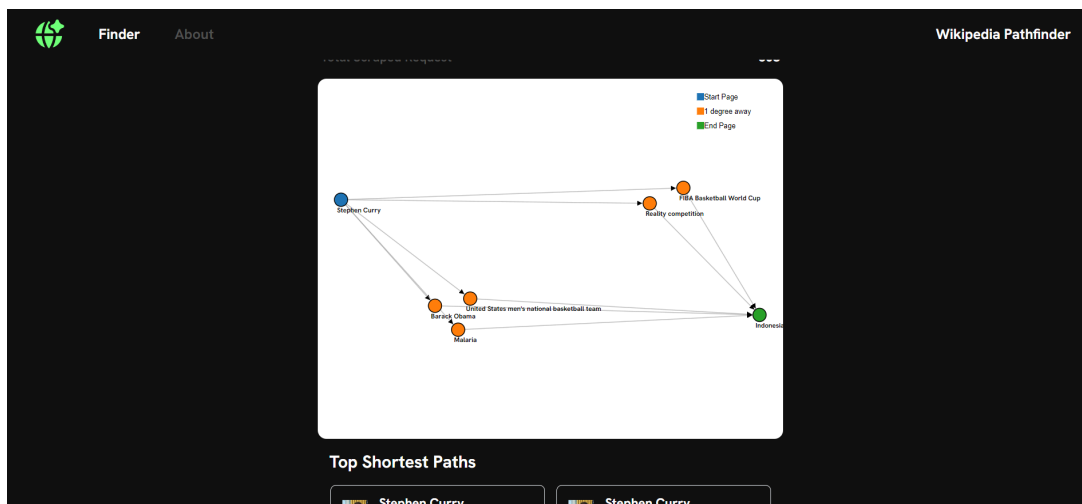
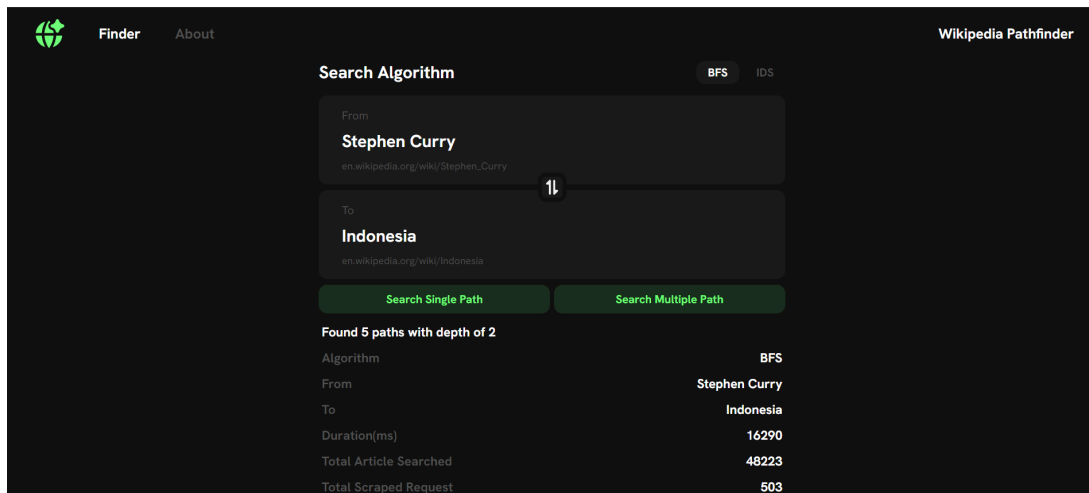


The screenshot shows the Wikipedia Pathfinder interface after caching. The search algorithm is set to IDS. The search path is from Stephen Curry to Indonesia. The results show 1 path with a depth of 2. The duration is 115 ms, and the total article searched is 125107. The total scraped request is 0.

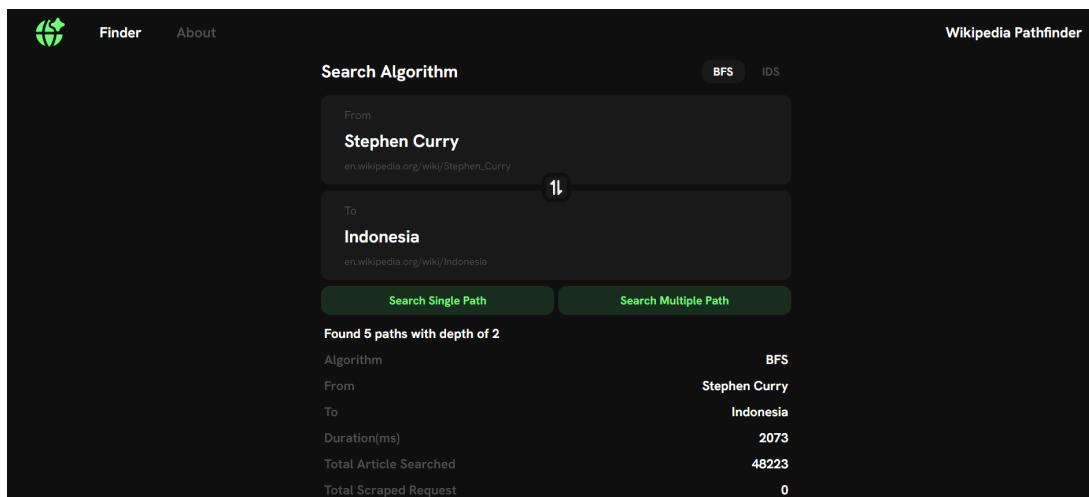
Found 1 paths with depth of 2	
Algorithm	IDS
From	Stephen Curry
To	Indonesia
Duration(ms)	115
Total Article Searched	125107
Total Scraped Request	0

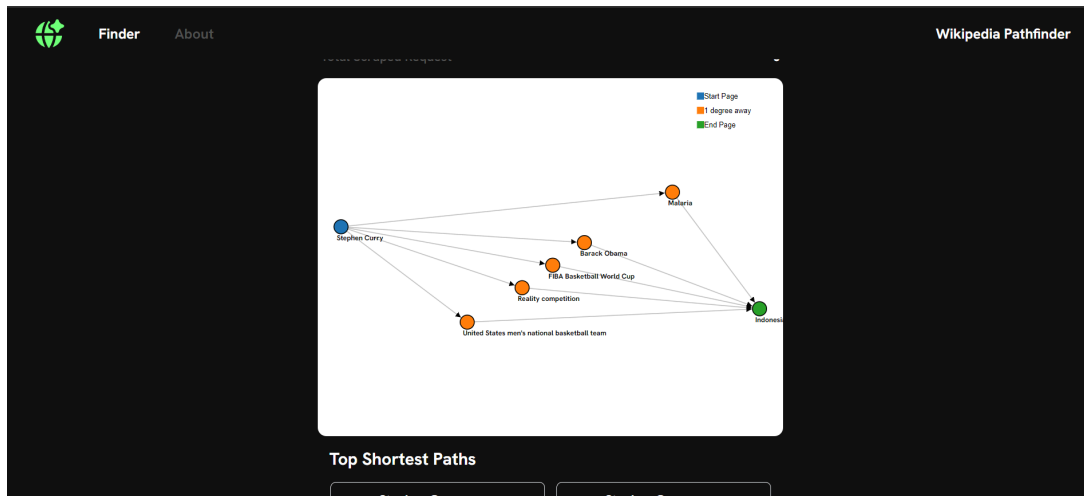
## BFS Multi path

Sebelum caching



Sesudah caching





## IDS Multipath

Sebelum caching

From

**Stephen Curry**
  
en.wikipedia.org/wiki/Stephen\_Curry

To

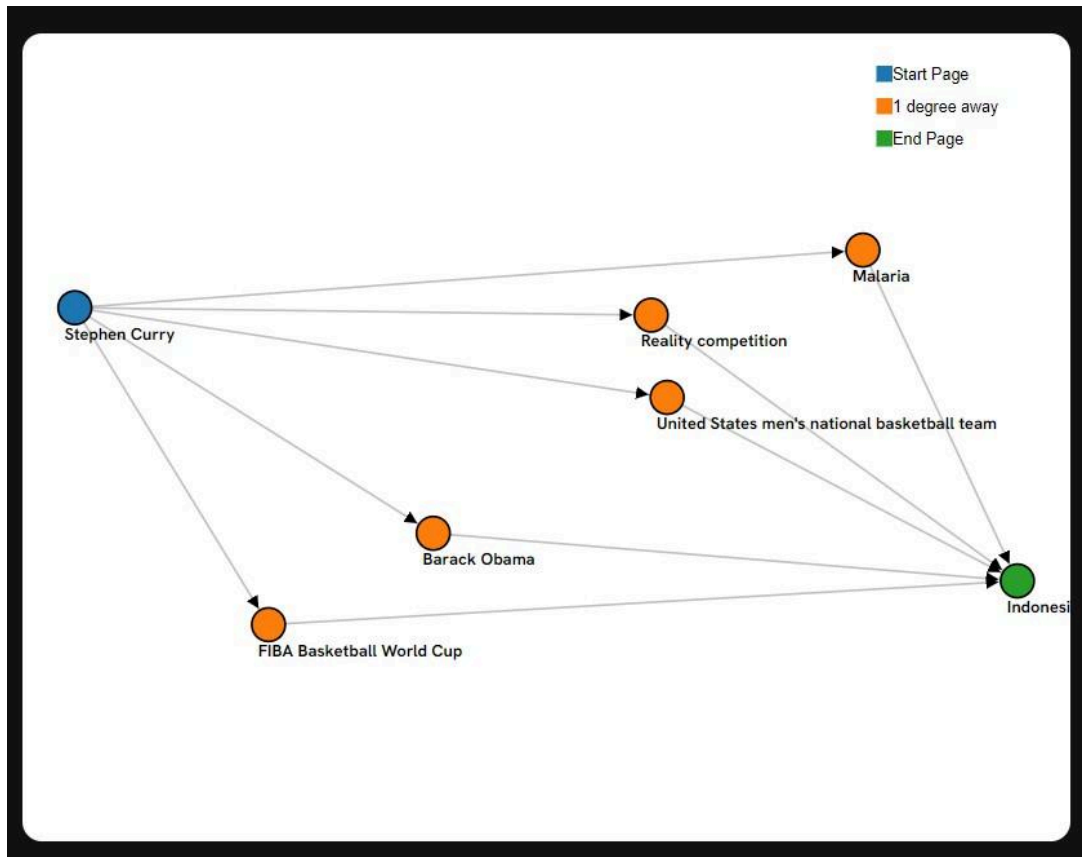
**Indonesia**
  
en.wikipedia.org/wiki/Indonesia

Search Single Path

Search Multiple Path

Found 5 paths with depth of 2

Algorithm	IDS
From	Stephen Curry
To	Indonesia
Duration(ms)	168009
Total Article Searched	46630
Total Scraped Request	461



Sesudah caching

Finder About Wikipedia Pathfinder

Search Algorithm BFS IDS

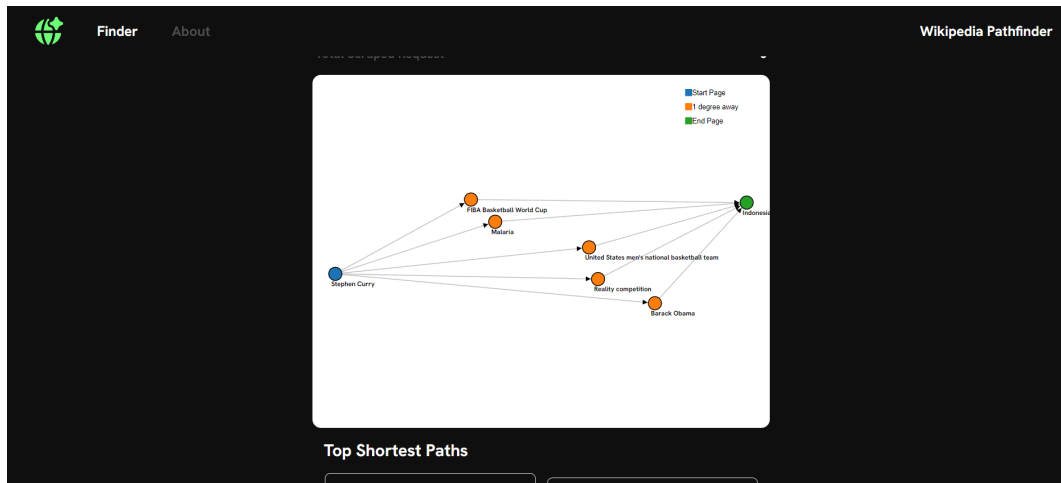
From  
**Stephen Curry**  
en.wikipedia.org/wiki/Stephen\_Curry

To  
**Indonesia**  
en.wikipedia.org/wiki/Indonesia

Search Single Path Search Multiple Path

Found 5 paths with depth of 2

Algorithm	IDS
From	Stephen Curry
To	Indonesia
Duration(ms)	160
Total Article Searched	48218
Total Scraped Request	0



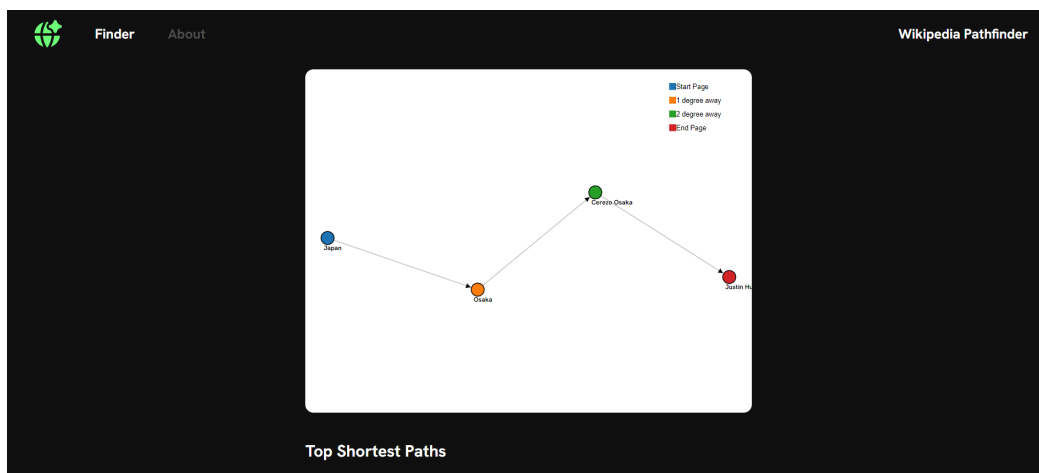
## 2. TestCase 2 (degree 3)

### BFS Single path

Sebelum caching

Wikipedia Pathfinder Search Algorithm interface. The search is performed using BFS (Breadth-First Search) from 'Japan' to 'Justin Hubner'. The results show 1 path found with a depth of 3.

Search Algorithm	
From	Japan <small>en.wikipedia.org/wiki/Japan</small>
To	Justin Hubner <small>en.wikipedia.org/wiki/Justin_Hubner</small>
<input type="button" value="Search Single Path"/> <input type="button" value="Search Multiple Path"/>	
Found 1 paths with depth of 3	
Algorithm	BFS
From	Japan
To	Justin Hubner
Duration(ms)	284523
Total Article Searched	671475
Total Scraped Request	11911



Sesudah caching

**Wikipedia Pathfinder**

Finder About

**Search Algorithm** BFS IDS

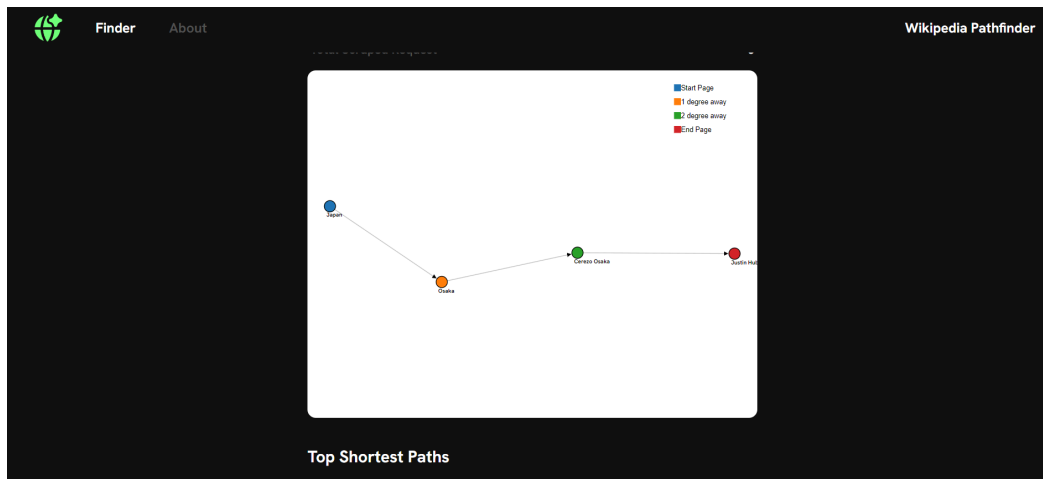
From  
**Japan**  
en.wikipedia.org/wiki/Japan

To  
**Justin Hubner**  
en.wikipedia.org/wiki/Justin\_Hubner

Search Single Path Search Multiple Path

Found 1 paths with depth of 3

Algorithm	BFS
From	Japan
To	Justin Hubner
Duration(ms)	3769
Total Article Searched	656308
Total Scraped Request	0



IDS Single path

Sebelum caching

**Wikipedia Pathfinder**

Finder About

**Search Algorithm** BFS IDS

From  
**Japan**  
en.wikipedia.org/wiki/Japan

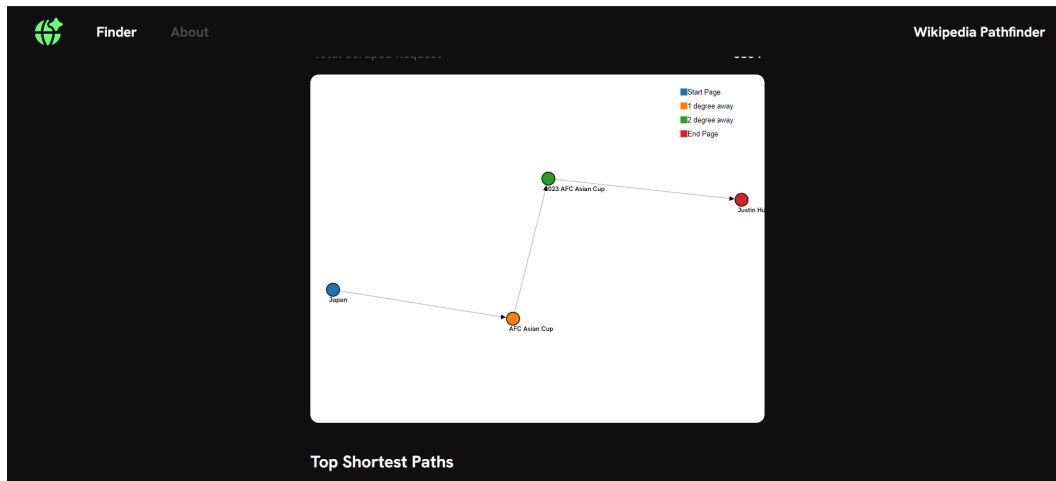
To  
**Justin Hubner**  
en.wikipedia.org/wiki/Justin\_Hubner

Search Single Path Search Multiple Path

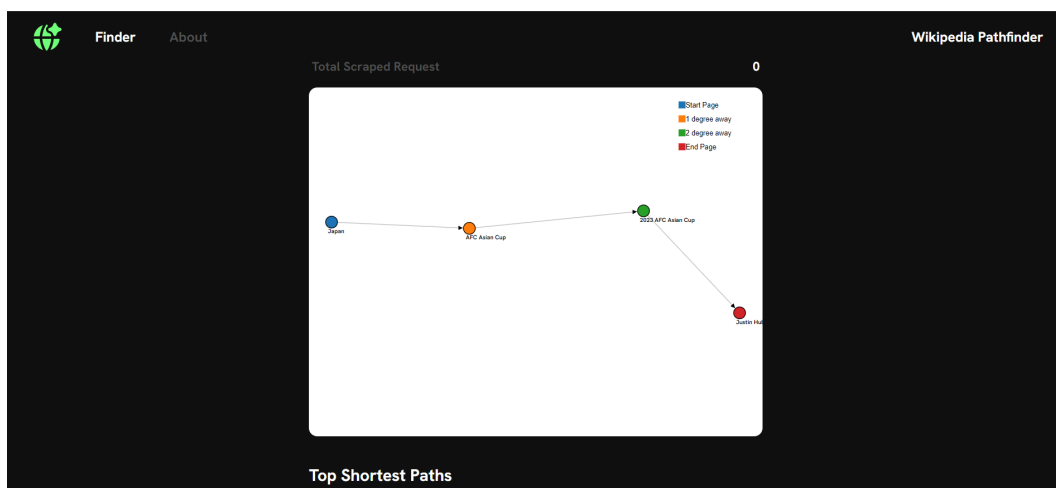
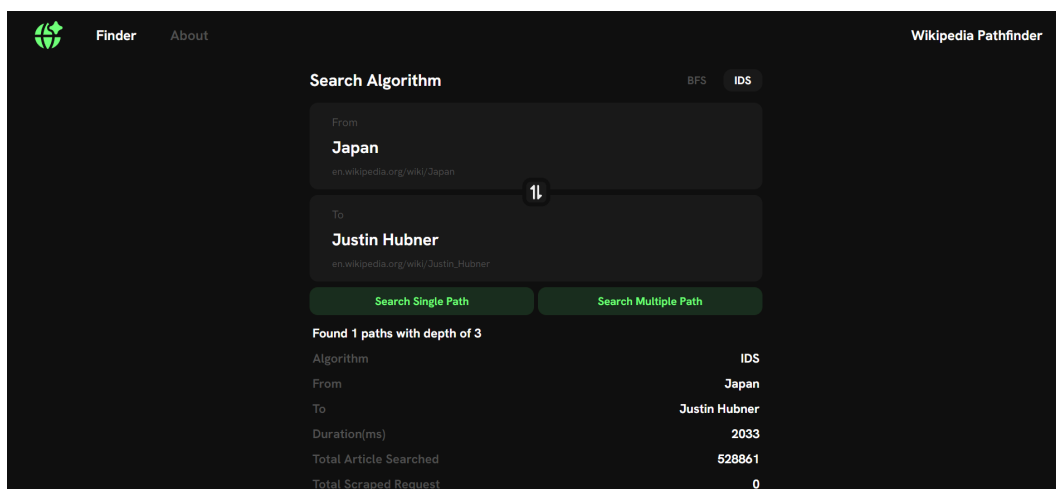
Found 1 paths with depth of 3

Algorithm	IDS
From	Japan
To	Justin Hubner
Duration(ms)	723949
Total Article Searched	528861
Total Scraped Request	6354





Sesudah caching



## BFS Multipath

Sebelum caching

Finder About Wikipedia Pathfinder

Search Algorithm **BFS** IDS

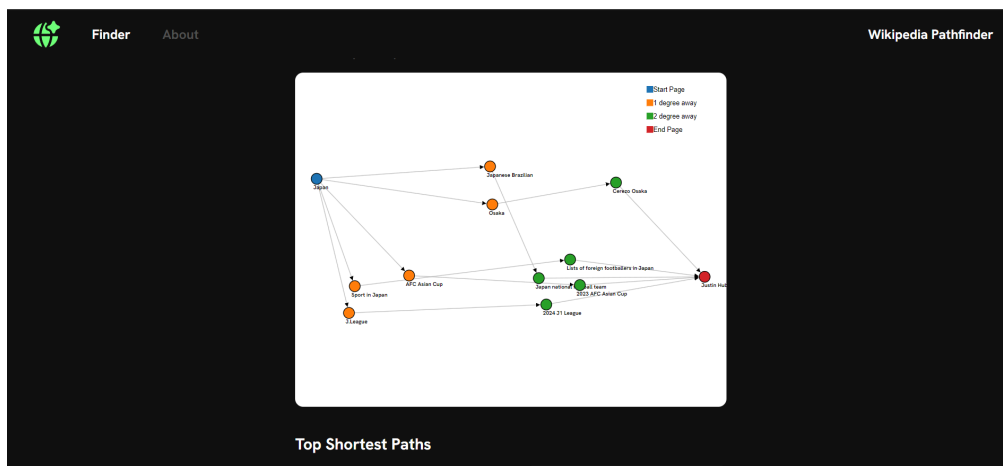
From  
**Japan**  
en.wikipedia.org/wiki/Japan

To  
**Justin Hubner**  
en.wikipedia.org/wiki/Justin\_Hubner

**Search Single Path** **Search Multiple Path**

Found 5 paths with depth of 3

Algorithm	BFS
From	Japan
To	Justin Hubner
Duration(ms)	1472054
Total Article Searched	2135923
Total Scraped Request	75712



Sesudah caching

Finder About Wikipedia Pathfinder

Search Algorithm **BFS** IDS

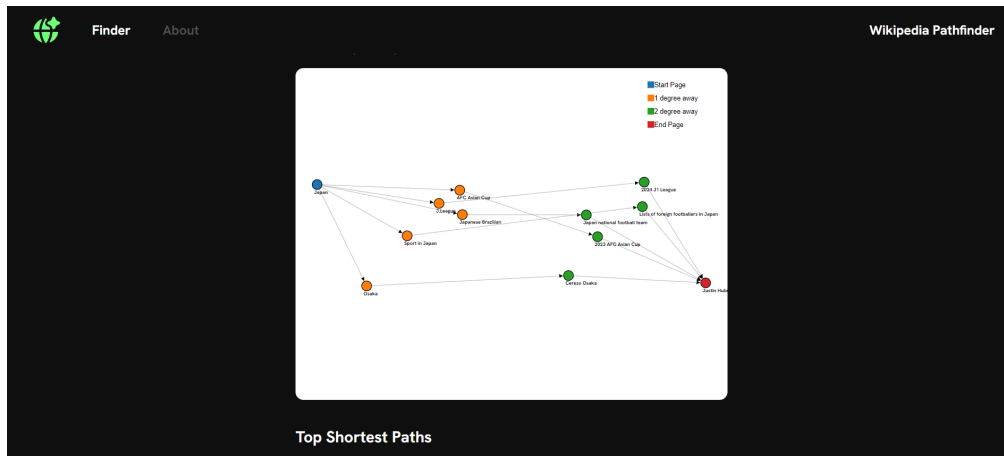
From  
**Japan**  
en.wikipedia.org/wiki/Japan

To  
**Justin Hubner**  
en.wikipedia.org/wiki/Justin\_Hubner

**Search Single Path** **Search Multiple Path**

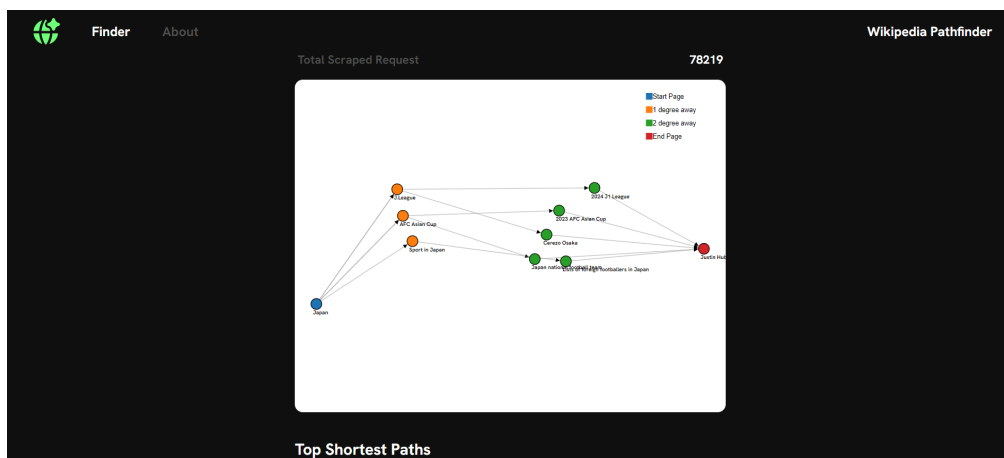
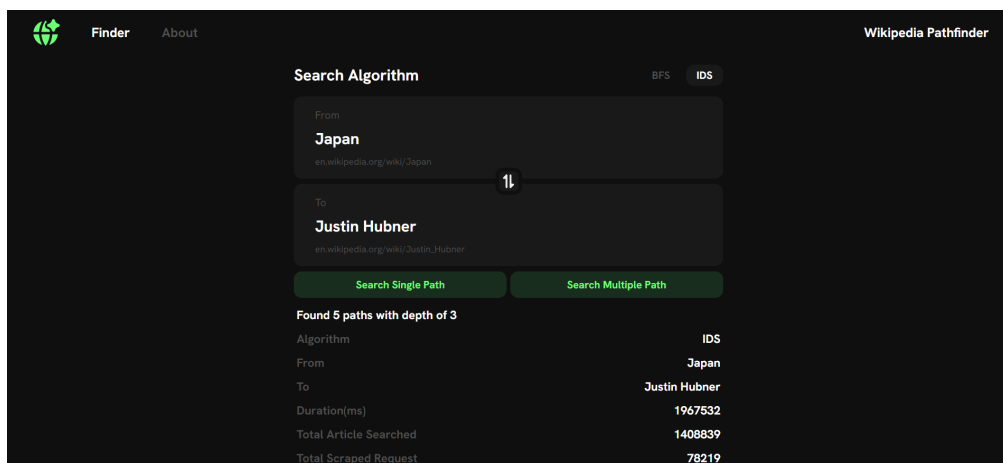
Found 5 paths with depth of 3

Algorithm	BFS
From	Japan
To	Justin Hubner
Duration(ms)	15797
Total Article Searched	2135935
Total Scraped Request	2

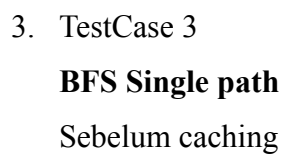


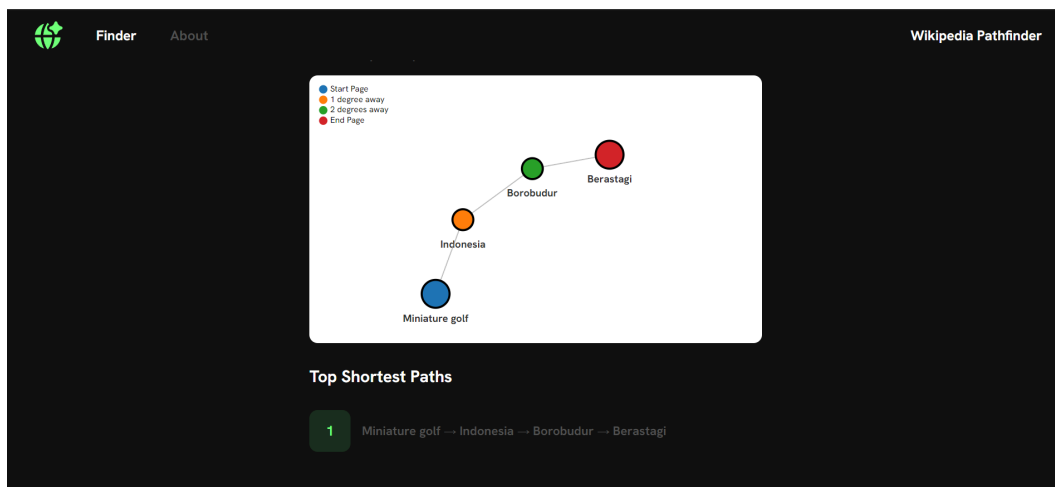
## IDS Multipath

Sebelum caching

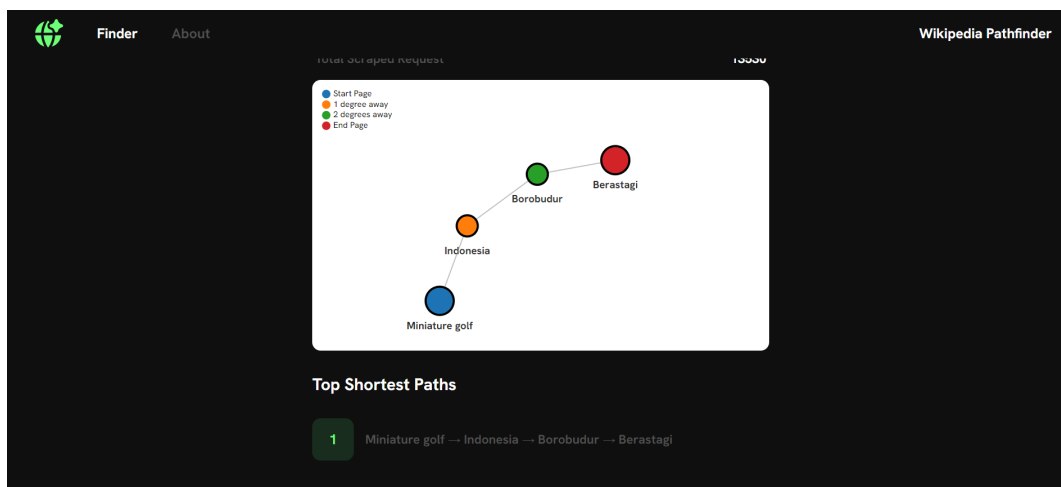
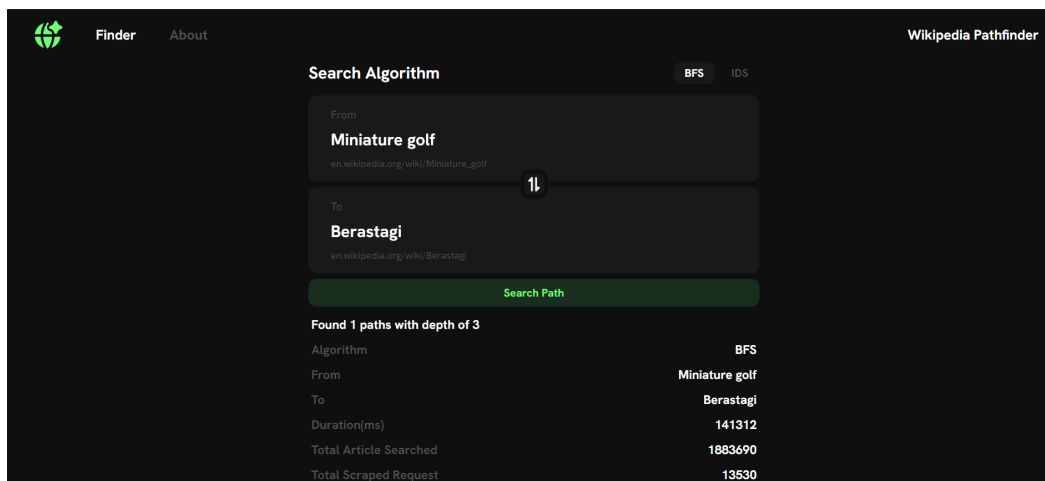


## Sesudah caching





Sesudah caching



## IDS Single path

Sebelum caching

Finder About Wikipedia Pathfinder

**Search Algorithm** BFS IDS

From  
**Miniature golf**  
en.wikipedia.org/wiki/Miniature\_golf

To  
**Berastagi**  
en.wikipedia.org/wiki/Berastagi

**Search Path**

Found 1 paths with depth of 3

Algorithm	BFS
From	Miniature golf
To	Berastagi
Duration(ms)	555106
Total Article Searched	2025115
Total Scraped Request	38932

Finder About Wikipedia Pathfinder

Legend:  
 ● Start Page  
 ● 1 degree away  
 ● 2 degrees away  
 ● End Page

**Top Shortest Paths**

1 Miniature golf → Indonesia → Borobudur → Berastagi

Sesudah caching

Finder About Wikipedia Pathfinder

**Search Algorithm** BFS IDS

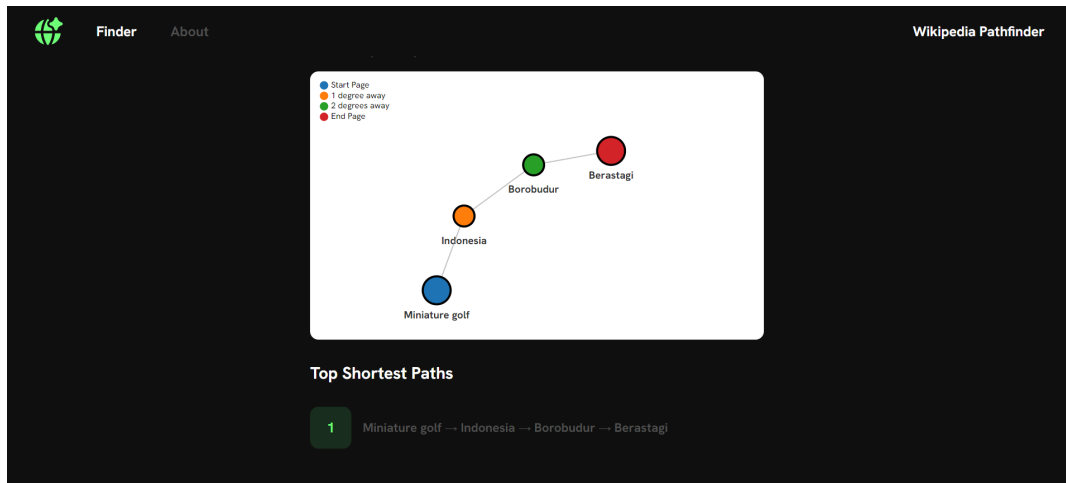
From  
**Miniature golf**  
en.wikipedia.org/wiki/Miniature\_golf

To  
**Berastagi**  
en.wikipedia.org/wiki/Berastagi

**Search Path**

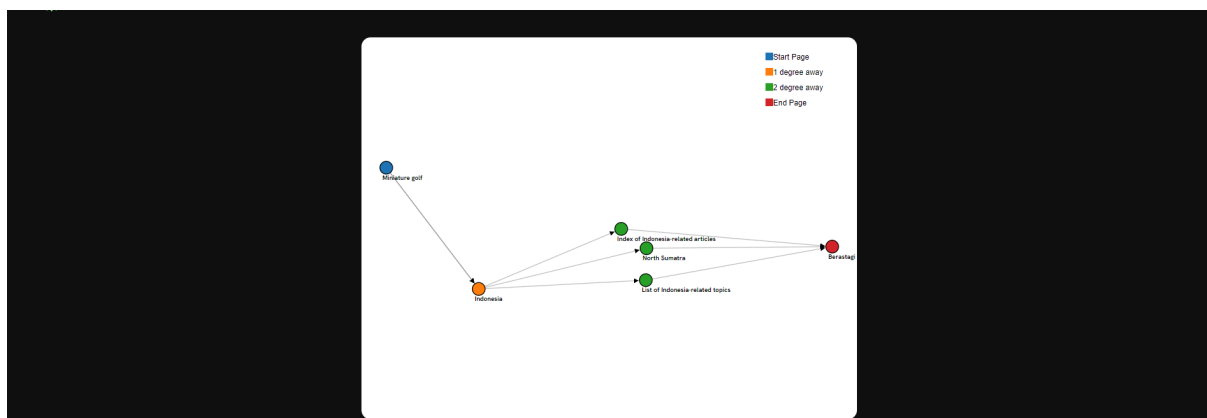
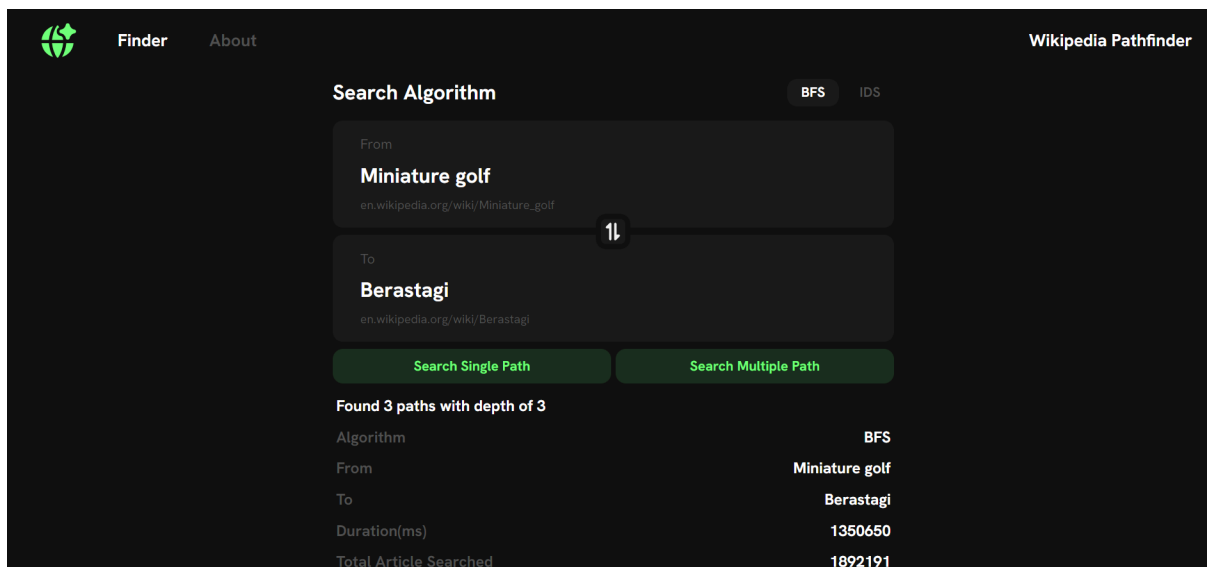
Found 1 paths with depth of 3

Algorithm	BFS
From	Miniature golf
To	Berastagi
Duration(ms)	59809
Total Article Searched	2058113
Total Scraped Request	3429



## BFS Multi path

Sebelum caching



Sesudah caching

**Wikipedia Pathfinder**

Finder About

**Search Algorithm** BFS IDS

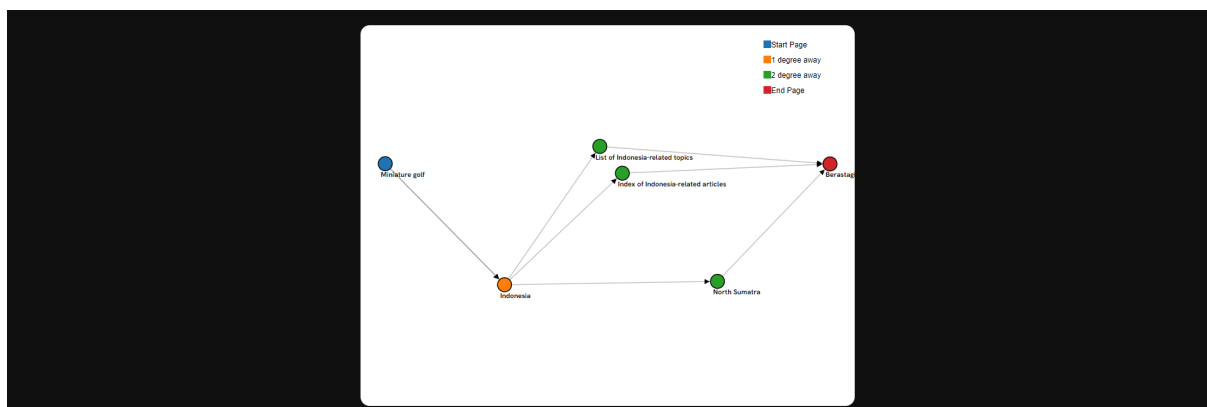
From  
**Miniature golf**  
en.wikipedia.org/wiki/Miniature\_golf

To  
**Berastagi**  
en.wikipedia.org/wiki/Berastagi

Search Single Path Search Multiple Path

Found 3 paths with depth of 3

Algorithm	BFS
From	Miniature golf
To	Berastagi
Duration(ms)	34860
Total Article Searched	1892191
Total Scraped Request	6



## IDS Multi path

Sebelum caching

**Wikipedia Pathfinder**

Finder About

**Search Algorithm** BFS IDS

From  
**Miniature golf**  
en.wikipedia.org/wiki/Miniature\_golf

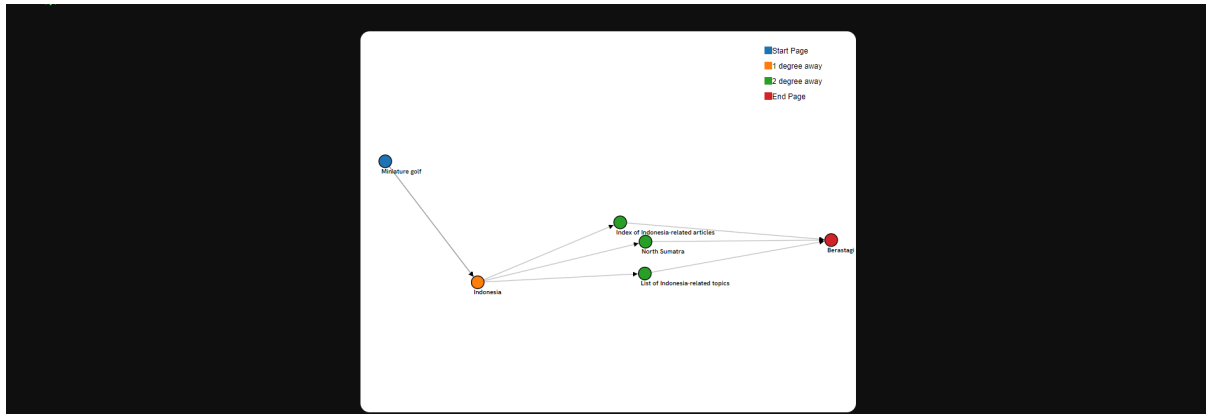
To  
**Berastagi**  
en.wikipedia.org/wiki/Berastagi

Search Single Path Search Multiple Path

Found 3 paths with depth of 3

Algorithm	IDS
From	Miniature golf
To	Berastagi
Duration(ms)	1537860
Total Article Searched	1973682
Total Scraped Request	49432





Sesudah caching

From  
**Miniature golf**  
en.wikipedia.org/wiki/Miniature\_golf

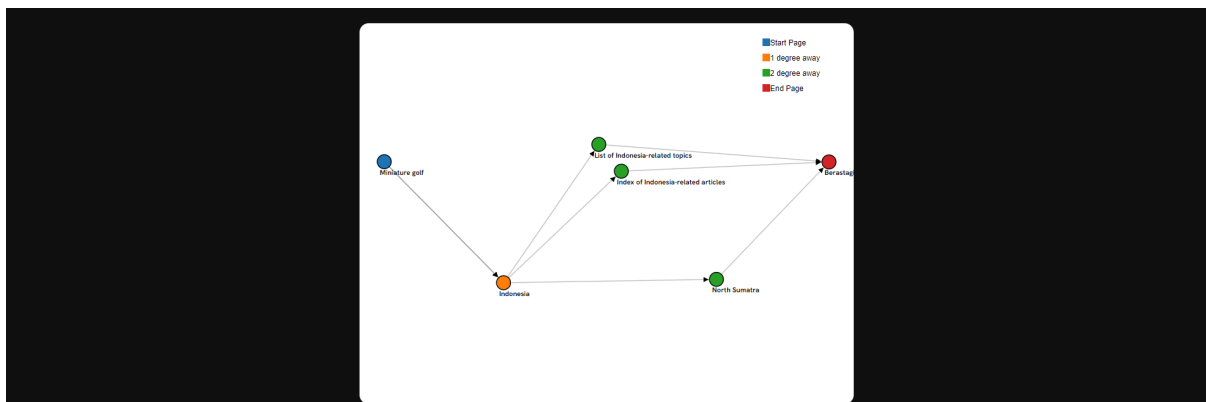
To  
**Berastagi**  
en.wikipedia.org/wiki/Berastagi

Search Single Path

Search Multiple Path

Found 3 paths with depth of 3

Algorithm	IDS
From	Miniature golf
To	Berastagi
Duration(ms)	26490
Total Article Searched	1892191
Total Scraped Request	6



#### D. Analisis Hasil Pengujian

Berdasarkan hasil pengujian, diketahui bahwa algoritma Iterative Deepening Search (IDS) membutuhkan pemeriksaan artikel yang lebih banyak dibandingkan dengan Breadth-First Search (BFS) pada pencarian artikel tujuan dari artikel awal yang sama. Sebelum caching diterapkan, IDS berjalan lebih lambat dari BFS karena beberapa alasan. Pertama, IDS melakukan pemeriksaan berulang pada setiap iterasi kedalaman, di mana setiap

kedalaman baru, pemeriksaan dimulai lagi dari kedalaman awal. Sementara itu, BFS tidak melakukan pemeriksaan ulang pada kedalaman yang telah selesai. Kedua, implementasi BFS telah menggunakan BFS asinkron yang mendukung paralelisme dan berjalan secara konkuren, sehingga prosesnya menjadi lebih cepat.

Namun, setelah penerapan caching, BFS malah berjalan lebih lambat dibandingkan IDS, yang pada analisis awal sepertinya tidak logis. Memang benar bahwa caching membuat kedua algoritma berjalan jauh lebih cepat dibandingkan sebelum caching diterapkan, karena pengurangan overhead yang terjadi selama proses scraping artikel. Namun, kecepatan BFS setelah caching menjadi lebih lambat dibanding IDS karena implementasi goroutine untuk konkurensi. Pada IDS yang bersifat sinkronus, ketika artikel tujuan ditemukan, algoritma langsung berhenti dan program segera terminasi. Sementara pada BFS, setelah artikel tujuan ditemukan, program masih harus memastikan bahwa semua goroutine yang telah dibangkitkan untuk BFS selesai atau diterminasi, sehingga menimbulkan overhead tambahan yang memperlambat terminasi program dibandingkan dengan durasi pencarian artikel itu sendiri.

## BAB V

### KESIMPULAN DAN SARAN

#### A. Kesimpulan

Dari tugas besar IF2211 Strategi Algoritma ini, telah diimplementasikan sebuah aplikasi berbasis website untuk menentukan lintasan terpendek yang perlu ditempuh dari suatu hipertaut artikel Wikipedia menuju artikel tujuan menggunakan algoritma BFS dan IDS. Aplikasi berbasis website tersebut dibangun dengan menggunakan *framework* React untuk frontend, dan bahasa pemrograman Go sebagai backend dengan spesifikasi fungsional dan tata cara penggunaan yang telah diuraikan pada bagian sebelumnya.

Berdasarkan implementasi dan analisis hasil pengujian yang telah diuraikan pada Bab IV, dapat diamati perbedaan performansi dari penentuan rute terpendek berdasarkan algoritma BFS dan IDS. Algoritma BFS yang diimplementasikan mengadopsi teknik pencarian asinkron dan paralelisme dengan memanfaatkan kakas Goroutine pada bahasa pemrograman Golang, sehingga memungkinkan proses pencarian menjadi lebih cepat dibandingkan dengan algoritma IDS yang berjalan secara sinkronus dan iteratif. Perbedaan performansi tersebut disebabkan oleh adanya pencarian berulang pada setiap iterasi kedalaman pada algoritma IDS, sehingga menyebabkan algoritma IDS berjalan lebih lambat dibandingkan BFS dengan detail penjelasan yang dapat diamati pada bagian Analisis Hasil Pengujian.

Selain itu, diimplementasikan pula teknik *caching* yang berfungsi untuk menyimpan hasil pencarian secara global, sehingga waktu yang diperlukan untuk melakukan untuk melakukan *scraping* berkurang secara cukup signifikan. Adapun pada implementasinya, teknik *caching* meningkatkan performansi dan efisiensi algoritma IDS secara signifikan, sehingga proses pencarian solusinya lebih cepat dibandingkan algoritma BFS yang menggunakan paralelisme. Hal ini disebabkan oleh adanya overhead tambahan yang diakibatkan oleh penggunaan goroutine pada BFS, sehingga terminasi program juga bergantung terhadap proses terminasi dari keseluruhan goroutine yang digunakan, meskipun artikel tujuan telah ditemukan, berbeda halnya dengan algoritma IDS yang secara sinkronus langsung berhenti apabila artikel tujuan telah ditemukan.

## B. Saran

Untuk mengoptimasi pencarian artikel menggunakan BFS dan IDS agar prosesnya lebih cepat, kita bisa memanfaatkan database yang telah dipersiapkan dengan data Wikipedia yang telah diproses terlebih dahulu. Ini melibatkan prescraping atau preprocessing dari Wikipedia data dump, sehingga mengurangi overhead dari scraping saat pencarian berlangsung. Kehadiran data yang siap pakai dalam database memungkinkan semua pencarian berjalan lebih lancar.

Selain itu, paralelisme dapat diimplementasikan pada IDS untuk meningkatkan kecepatan. Meskipun paralelisme belum diimplementasikan pada program yang ada, integrasi ini dapat mempercepat pencarian secara signifikan dengan memungkinkan pencarian simultan pada berbagai cabang.

Untuk mempercepat BFS, penggunaan teknik double-ended BFS dapat diadopsi. Teknik ini melibatkan inisiasi BFS dari kedua artikel awal dan tujuan secara simultan. Setiap node atau artikel yang terhubung dengan artikel tujuan dianggap relevan dan berpotensi mempercepat pertemuan kedua pencarian. Jika BFS yang dimulai dari artikel awal bertemu dengan artikel yang telah ditemukan oleh BFS dari artikel tujuan, algoritma dapat melanjutkan dari titik pertemuan tersebut secara heuristik (tidak langsung ditanyakan artikel tujuan ditemukan karena wikipedia bersifat one-way directed), dengan asumsi bahwa ini akan mengarah langsung ke artikel tujuan.

## LAMPIRAN

### **Pranala repository**

[https://github.com/DerwinRustanly/Tubes2\\_JuBender](https://github.com/DerwinRustanly/Tubes2_JuBender)

### **Pranala video *youtube***

<https://youtu.be/PWLnlVCZGw0?feature=shared>

### **Pranala cache**

<https://bit.ly/cacheJuBender>

## 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>