

LAPORAN TUGAS BESAR 02

IF2211 STRATEGI ALGORITMA

PEMANFAATAN ALGORITMA IDS DAN BFS DALAM PERMAINAN WIKIRACE

Kelompok 26 : GoPat



Disusun oleh:

Filbert 13522021

Ivan Hendrawan Tan 13522111

Mesach Harmasendro 13522117

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA (STEI)
INSTITUT TEKNOLOGI BANDUNG**

2024

DAFTAR ISI

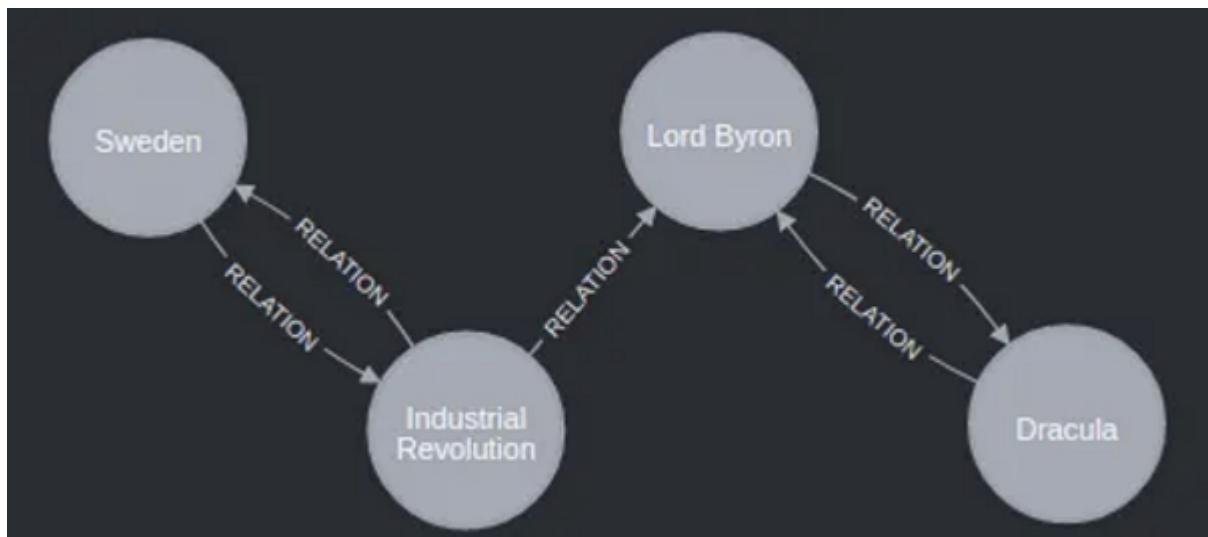
DAFTAR ISI.....	1
BAB I DESKRIPSI MASALAH.....	3
BAB II LANDASAN TEORI.....	4
2.1 Penjelajahan Graf.....	4
2.2 BFS.....	4
2.3 IDS.....	5
2.4 Pembangunan Aplikasi Web.....	6
2.4.1 Pengenalan Teknologi.....	6
2.4.2 Next.js sebagai Framework Front-End.....	6
2.4.3 Tailwind CSS untuk Styling.....	7
2.4.4 Backend dengan Golang.....	7
2.4.5 PostgreSQL untuk Database.....	7
2.4.6 Dockerfile.....	7
2.4.7 Arsitektur dan Interaksi.....	8
BAB III ANALISIS PEMECAHAN MASALAH.....	9
3.1 Langkah Pemecahan Masalah.....	9
3.1.1 Pendefinisian Masalah.....	9
3.1.2 Penetapan Desain Solusi.....	9
3.1.3 Integrasi dengan Backend.....	9
3.1.4 Pemrosesan dan pengoptimalan program.....	10
3.1.5 Caching.....	10
3.1.6 Respon ke frontend.....	11
3.1.7 Evaluasi dan Pengujian.....	11
3.2 Proses Pemetaan Masalah.....	11
3.2.1 Pemetaan Masalah menjadi Elemen Untuk Algoritma BFS.....	11
3.2.2 Pemetaan Masalah menjadi Elemen Untuk Algoritma IDS.....	13
3.3 Fitur Fungsional dan Arsitektur Aplikasi Web.....	14
3.4 Ilustrasi Kasus.....	17
BAB IV IMPLEMENTASI DAN PENGUJIAN.....	19
4.1 Implementasi dalam Pseudocode.....	19
4.2 Spesifikasi Teknis Program dan Struktur Data Program.....	33
4.2.1 Algoritma BFS.....	33
4.2.2 Algoritma IDS.....	35
4.3 Tata Cara Penggunaan Program.....	36
4.4 Hasil Pengujian.....	39
1. Test Case 1 (2 degrees).....	39
2. Test Case 2 (3 degrees).....	42
3. Test Case 3 (3 degrees).....	45
4. Test Case 4 (4 degrees).....	47
4.5 Analisis Pengujian.....	50
BAB V PENUTUP.....	52

5.1 Kesimpulan.....	52
5.2 Saran.....	53
DAFTAR PUSTAKA.....	54
LAMPIRAN.....	55

BAB I

DESKRIPSI MASALAH

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



Gambar 1.1 Ilustrasi Graf pada Permainan WikiRace

Program yang akan dikembangkan menggunakan bahasa Go dan menggunakan dua algoritma pencarian, yaitu Iterative Deepening Search (IDS) dan Breadth-First Search (BFS), untuk mendapatkan solusi dalam permainan WikiRace. Pengguna dapat memilih jenis algoritma yang diinginkan serta memasukkan judul artikel awal dan tujuan melalui antarmuka web. Output yang dihasilkan oleh program ini mencakup jumlah artikel yang diperiksa, jumlah artikel yang dilalui, rute penjelajahan dari artikel awal hingga artikel tujuan, serta waktu pencarian yang diukur dalam milidetik. Program ini dirancang untuk mengeluarkan hanya satu rute terpendek, namun tak terbatas bagi yang mengerjakan bonus. Program ini juga dirancang untuk menemukan rute terpendek dalam waktu kurang dari lima menit untuk setiap pencarian. Program ini berbasis web sehingga memiliki komponen front-end dan back-end yang dapat disimpan menjadi satu atau dipisah.

BAB II

LANDASAN TEORI

2.1 Penjelajahan Graf

Graf adalah sebuah metode untuk merepresentasikan objek-objek diskrit dan hubungan antara objek-objek tersebut. Graf G terdiri dari pasangan himpunan (V, E) yang ditulis $G = (V, E)$ dengan V adalah himpunan tak kosong dari simpul (*vertices*) dan E adalah himpunan sisi (*edges*) yang menghubungkan sepasang simpul.

Penjelajahan graf adalah sebuah algoritma yang mengunjungi simpul-simpul di dalam graf dengan cara yang sistematis. Penjelajahan graf juga dapat didefinisikan sebagai proses pencarian solusi sebuah permasalahan yang direpresentasikan dalam bentuk graf. Secara umum, terdapat dua metode yang paling sering digunakan untuk melakukan penjelajahan graf, yaitu metode pencarian melebar (BFS, *Breadth-First Search*) dan metode pencarian mendalam (DFS, *Depth-First Search*).

Dalam melakukan pencarian solusi, terdapat dua cara untuk menjalankannya yaitu pencarian tanpa informasi (*uninformed* atau *blind search*) dan pencarian dengan informasi (*informed search*). Sesuai dengan namanya, pencarian tanpa informasi tidak menyediakan informasi tambahan untuk melakukan pencarian. Pencarian ini mencakup algoritma seperti DFS, BFS, *Depth Limited Search*, *Iterative Deepening Search*, dan *Uniform Cost Search*. Pencarian dengan informasi melakukan pencarian dengan berbasis heuristik dengan tujuan mendapatkan solusi yang paling optimal, contoh algoritma pencarian dengan informasi adalah *Best First Search* dan A^* .

2.2 BFS

Breadth-First Search (BFS), atau pencarian melebar, merupakan salah satu algoritma pencarian graf yang paling umum dan banyak digunakan. Dengan memulai dari simpul sumber, BFS secara sistematis mengeksplorasi setiap simpul di graf, melebarkan eksplorasinya secara berlapis-lapis dari simpul yang paling dekat dengan sumber ke simpul yang lebih jauh. Eksplorasi ini menggunakan struktur data antrian untuk menjaga agar prosesnya tetap pada tingkat kedalaman yang sama sebelum bergerak ke kedalaman berikutnya. Setiap simpul yang dikeluarkan dari antrian dijadikan titik pusat untuk menjelajahi tetangganya, yang belum dikunjungi. Setelah semua tetangga dari simpul tersebut dikunjungi, mereka ditambahkan ke antrian,

memastikan bahwa proses eksplorasi berlangsung secara merata dan bertahap melintasi setiap tingkat kedalaman.

Keunggulan BFS terletak pada kemampuannya untuk menemukan jalur terpendek dalam graf tak berbobot, karena ia mengunjungi simpul-simpul dalam urutan jarak meningkat dari titik awal. Algoritma ini juga sangat efektif dalam situasi di mana solusi diharapkan tidak jauh dari simpul awal, karena secara bertahap melebarkan pencarian dari titik tersebut. Namun, tantangan utama BFS adalah konsumsi memori yang tinggi ketika menjelajahi graf dengan faktor percabangan besar, karena setiap simpul pada satu kedalaman harus disimpan dalam memori sebelum bergerak ke kedalaman berikutnya.

Dalam praktiknya, BFS banyak dimanfaatkan dalam berbagai aplikasi, dari penentuan jalur terpendek dalam jaringan transportasi hingga navigasi algoritma dalam game dan aplikasi jaringan sosial. Keefektifan dan kelengkapan BFS dalam menjelajahi semua elemen graf membuatnya menjadi pilihan populer dalam perencanaan dan analisis jaringan, menunjukkan bagaimana eksplorasi yang terstruktur dan metodis bisa mengungkap solusi yang optimal dan komprehensif.

2.3 IDS

Iterative Deepening Search (IDS) adalah algoritma pencarian yang dirancang untuk menggabungkan keunggulan dari kedua algoritma pencarian yang umum, yaitu *Depth-First Search* (DFS) dan *Breadth-First Search* (BFS). IDS melakukan ini dengan mengadopsi pendekatan pencarian yang mendalam dari DFS, sambil mengimplementasikan kontrol batasan kedalaman yang mirip dengan BFS, membuatnya sangat efisien dalam hal penggunaan memori.

Algoritma IDS bekerja dengan secara iteratif meningkatkan batas kedalaman untuk pencarian DFS. Pada iterasi pertama, algoritma melakukan pencarian seperti DFS biasa namun hanya hingga kedalaman yang sangat terbatas (misalnya, satu level kedalaman). Jika solusi tidak ditemukan, kedalaman batasan ditingkatkan, dan pencarian dimulai ulang dari simpul akar. Proses ini berulang hingga solusi ditemukan atau semua simpul telah ditelusuri.

Keunggulan utama IDS adalah efisiensinya dalam hal penggunaan memori. Seperti DFS, IDS hanya memerlukan ruang untuk menyimpan jalur dari akar ke simpul saat ini dan saudara kandungnya, yang jauh lebih sedikit dibandingkan dengan BFS yang memerlukan penyimpanan semua simpul pada level kedalaman tertentu. Ini

membuat IDS sangat cocok untuk pencarian di ruang keadaan yang besar di mana penggunaan memori menjadi pertimbangan kritis.

IDS juga sangat efektif dalam menemukan solusi terpendek dengan lebih cepat dalam pohon atau graf yang luas dan dalam. Karena setiap iterasi IDS sebenarnya adalah pencarian DFS penuh hingga kedalaman tertentu, ia mencapai keseluruhan simpul lebih cepat daripada BFS dalam kebanyakan kasus, terutama di graf yang luas di mana BFS mungkin memperlambat secara signifikan karena eksplorasi level demi level.

Secara konseptual, IDS dapat dipandang sebagai hibrida DFS dan BFS. Algoritma ini menggunakan ruang yang lebih kecil seperti DFS tetapi mencoba untuk mencapai kelengkapan dan optimalitas dari BFS dengan mengulangi eksplorasi hingga kedalaman yang semakin meningkat. Dengan demikian, IDS menggabungkan keuntungan utama dari kedua algoritma tersebut, menjadikannya pilihan yang baik untuk berbagai masalah pencarian di mana kedalaman solusi tidak diketahui atau berpotensi dalam, dan penggunaan memori adalah batasan penting.

2.4 Pembangunan Aplikasi Web

2.4.1 Pengenalan Teknologi

Dalam tugas besar ini, dikembangkan sebuah aplikasi web yang mengintegrasikan teknologi front-end dan back-end untuk menyediakan pengalaman pengguna yang cepat dan responsif dalam memainkan WikiRace. Aplikasi ini menggunakan framework Next.js untuk front-end dan bahasa pemrograman Golang untuk back-end, dengan styling yang dikelola oleh Tailwind CSS.

2.4.2 Next.js sebagai Framework Front-End

Next.js adalah framework JavaScript yang memungkinkan pengembang untuk membangun aplikasi web yang dinamis dan statis dengan efisien. Framework ini mendukung fitur seperti *Server-Side Rendering* dan *Static Site Generation*, yang meningkatkan performa dan SEO aplikasi. Dalam proyek ini, Next.js digunakan untuk membangun tampilan antarmuka pengguna yang interaktif, memanfaatkan sistem routing bawaan dan optimasi gambar otomatis untuk mempercepat waktu muat halaman.

2.4.3 Tailwind CSS untuk Styling

Tailwind CSS adalah sebuah framework CSS yang sangat dapat dikustomisasi dan memungkinkan pembuatan desain yang responsif dengan cepat tanpa meninggalkan file CSS. Dalam aplikasi ini, Tailwind CSS digunakan untuk mendesain antarmuka dengan pendekatan utility-first, memungkinkan kami untuk membangun desain yang konsisten dan mudah diadaptasi di berbagai perangkat dan ukuran layar.

2.4.4 Backend dengan Golang

Golang, atau Go, adalah bahasa pemrograman yang dikembangkan oleh Google, terkenal dengan performa yang tinggi dan efisiensi dalam pengelolaan konkurensi. Di backend, Golang digunakan untuk menangani logika aplikasi, interaksi dengan database, dan integrasi dengan sistem lainnya. Umumnya, framework seperti Gin sering digunakan dalam pengembangan aplikasi Go untuk mempermudah pengelolaan routing, middleware, dan fitur-fitur lainnya. Gin memungkinkan pengembang untuk dengan cepat membangun API web dengan kinerja yang tinggi.

2.4.5 PostgreSQL untuk Database

PostgreSQL adalah sistem manajemen basis data relasional (RDBMS) yang bersifat open source. Ini dikembangkan dengan dukungan penuh untuk integritas data dan fitur SQL. PostgreSQL mendukung fitur-fitur lanjutan seperti transaksi ACID (Atomicity, Consistency, Isolation, Durability), query kompleks, jenis data asing, index yang dapat disesuaikan, dan ekstensi yang bisa ditambahkan oleh pengguna. Kecepatan dan kemampuan menyesuaikan PostgreSQL menjadikannya pilihan yang cocok untuk proyek ini yang membutuhkan pengelolaan database yang efisien dan fleksibel.

2.4.6 Dockerfile

Dockerfile adalah skrip yang digunakan untuk membuat images di Docker, sebuah platform yang memungkinkan virtualisasi pada tingkat sistem operasi, dikenal sebagai kontainerisasi. Dockerfile berisi serangkaian instruksi yang menentukan bagaimana lingkungan dari suatu aplikasi harus dibangun. Setiap instruksi dalam Dockerfile menambahkan lapisan ke image, dan setiap lapisan disimpan dan di cache untuk digunakan di masa depan.

2.4.7 Arsitektur dan Interaksi

Aplikasi ini dirancang dengan arsitektur yang memisahkan dengan jelas antara logika front-end dan back-end, dimana front-end mengirimkan permintaan ke back-end melalui API yang dikembangkan di Golang. Penggunaan JSON sebagai format pertukaran data memungkinkan komunikasi yang efisien dan cepat antar dua sisi aplikasi dengan memanfaatkan HTTP sebagai protokol komunikasi.

BAB III

ANALISIS PEMECAHAN MASALAH

3.1 Langkah Pemecahan Masalah

3.1.1 Pendefinisian Masalah

Aplikasi web ini dikembangkan untuk memudahkan pengguna untuk mencari rute terpendek yang menghubungkan dua halaman wikipedia. Pengguna akan memberikan masukan berupa halaman sumber dan tujuan. Pengguna dapat memilih antara algoritma BFS atau IDS untuk melakukan pencarian. Tujuan dari pembuatan aplikasi ini adalah untuk memberikan solusi rute terpendek yang tepat, cepat, akurat serta menampilkan rute yang ditemukan agar pengguna tidak perlu mencari link secara manual yang akan sangat memakan waktu.

3.1.2 Penetapan Desain Solusi

Setelah masalah didefinisikan, kami menetapkan framework untuk frontend dan backend aplikasi web ini. Untuk frontend, kami menggunakan Next.js dan Tailwind CSS, yang mudah digunakan dan mendukung pengembangan. Untuk backend, kami memilih framework Gin dengan bahasa Go. Sesudah memilih framework, dibuatlah sebuah desain yang sangat user friendly. Di dalam web page kita terdapat sebuah form yang akan meminta input pengguna untuk judul artikel awal dan akhir wikipedia. Saat melakukan pengisian akan terdapat rekomendasi yang sesuai penginputan yang berupa dropdown. Kemudian, pengguna harus memilih algoritma yang digunakan. Selain itu, terdapat tombol Go untuk melakukan pencarian rute jalur terpendek. Informasi mengenai berapa *degree* atau *depth*, waktu yang dibutuhkan dan juga rute jalur terpendek nya akan ditampilkan. Untuk visualisasi rute, digunakan *library javascript* d3.js

3.1.3 Integrasi dengan Backend

Saat tombol "Go" ditekan, sistem melakukan komunikasi antara frontend dan backend melalui *fetch API* untuk mengirimkan permintaan *HTTP*. Data yang dikirim meliputi informasi mengenai halaman awal dan halaman tujuan, serta algoritma pencarian yang telah dipilih oleh pengguna. Di sisi backend, terdapat endpoint khusus yang siap menerima dan memproses

permintaan tersebut berdasarkan algoritma yang dipilih. Proses ini memungkinkan sistem untuk mengolah dan mengembalikan hasil pencarian yang sesuai dengan permintaan pengguna.

3.1.4 Pemrosesan dan pengoptimalan program

Dalam proses berikutnya, data akan diolah menggunakan algoritma pencarian yang telah dipilih dan diikuti dengan pengambilan data atau scraping dari halaman Wikipedia. Hasil dari pengambilan data ini kemudian disimpan dalam struktur data khusus. Selanjutnya, sistem melakukan verifikasi data untuk menentukan adanya kesesuaian dengan judul yang ditargetkan. Jika ditemukan kesesuaian, sistem akan memberikan jalur yang dapat diikuti untuk mencapai judul tersebut. Proses pencarian ini juga mengimplementasi parallelism. Menggunakan teknik parallelism, pencarian dikelola oleh beberapa Goroutine yang beroperasi secara paralel dan konkuren, memungkinkan sistem untuk memproses pencarian secara lebih efisien dan cepat.

3.1.5 Caching

Setelah melakukan proses web scraping, Setiap link dari halaman Wikipedia ini kemudian akan disimpan atau di-cache. Penyimpanan ini menggunakan sistem database PostgreSQL. Keberadaan cache ini sangat membantu dalam mempercepat proses pencarian kedepannya. Ketika ada permintaan untuk mengakses halaman Wikipedia yang sama kedepannya, sistem akan terlebih dahulu memeriksa cache. Jika data halaman tersebut sudah tersimpan di cache, sistem akan menggunakan data tersebut daripada harus melakukan scraping ulang. Ini menghindari pemborosan waktu karena scraping membutuhkan waktu dan bisa berpotensi mendapatkan penolakan permintaan oleh server Wikipedia akibat terlalu banyak request. Sebenarnya penyimpanan dalam database lebih pantas dikatakan sebagai dataset dibandingkan caching. Namun karena jumlah dataset yang masih cukup sedikit dibandingkan keseluruhan jumlah artikel maka masih tidak masalah jika penyimpanan ini dianggap sebagai caching.

3.1.6 Respon ke frontend

Setelah data diolah dan pencarian rute terpendek selesai, sistem mengirimkan respons kembali ke frontend. Setelah itu, akan ditampilkan informasi berupa teks dan juga hasil visualisasi dari rute terpendeknya dalam bentuk grafik. Grafik tersebut dapat dipindah-pindah dan di zoom in atau zoom out sesuai dengan kebutuhan pengguna.

3.1.7 Evaluasi dan Pengujian

Setelah program telah selesai dirancang, dilakukan pengujian terhadap program dengan mencoba beberapa testcase lalu mengecek hasil yang diperoleh dan dibandingkan dengan referensi web lain yaitu Six Degrees of Wikipedia. Dilakukan revisi dan pengujian berkali-kali agar hasil rute terpendek dapat dicari dalam waktu yang cepat.

3.2 Proses Pemetaan Masalah

3.2.1 Pemetaan Masalah menjadi Elemen Untuk Algoritma BFS

Elemen:

- Node Awal : Artikel awal Wikipedia.
- Node Tujuan : Artikel tujuan Wikipedia.
- Tujuan : Mencari jalur terpendek dari artikel awal ke artikel tujuan melalui link yang ada pada setiap artikel.
- Queue: Menyimpan artikel-artikel yang perlu dieksplorasi lebih lanjut. Dalam kasus multi path queue tidak hanya menyimpan node yang perlu dieksplorasi namun juga menyimpan data path lengkap menuju ke node tersebut.
- Visited Map: Mencatat semua artikel yang sudah dikunjungi agar tidak terjadi pengulangan.
- Parent Map: Mencatat parent dari setiap node yang ada (digunakan dalam kasus single path).

Proses:

Untuk mempercepat proses BFS yang dilakukan bersamaan dengan proses scrapping maka akan dilakukan proses paralelisme dalam pengelolaan queue.

1. Pada tahap pertama, akan dicek apakah artikel awal sama dengan artikel tujuan atau tidak jika sama maka path akan langsung ditemukan dan jika tidak maka artikel awal akan dimasukkan ke dalam queue. Lalu status visitednya pada map akan diubah menjadi true.
2. Artikel awal kemudian akan diambil dari dalam queue dan kemudian akan dicari semua hyperlink yang ada pada artikel tersebut. Hyperlink-hyperlink tersebut kemudian akan dicek jika merupakan tujuan maka akan langsung keluar namun jika tidak maka link artikel tersebut akan dimasukkan ke dalam queue dan status visitednya akan diubah.
3. Pada tahap ini ukuran queue sudah cukup besar sehingga bisa mulai dibagi sesuai dengan jumlah worker. Pertama-tama akan dihitung jumlah elemen pada queue yang akan diproses oleh masing-masing worker. Kemudian queue akan dibagi berdasarkan data tersebut dan queue yang sudah terbagi-bagi tersebut akan dikirimkan pada worker dan akan diproses. Proses pembagian queue ini akan dilakukan pada setiap kedalaman sehingga akan dipastikan bahwa proses yang berlangsung secara paralel akan lebih aman karena berada pada kedalaman yang sama.
4. Pada setiap worker link artikel akan diambil satu persatu dari dalam subqueue. Kemudian link itu akan dieksplorasi dan dicari semua hyperlink yang terdapat didalamnya. Semua hyperlink yang sudah didapatkan tadi kemudian akan dicek apakah sama dengan link tujuan atau tidak jika sama maka path ditemukan dan jika tidak sama dan link tersebut belum pernah dikunjungi maka link tersebut akan dimasukkan ke dalam queue kembali dan status visitednya akan diubah serta data parent dari link tersebut akan disimpan pada parent map.
5. Proses pada no 4 akan berlangsung pada semua worker hingga path ditemukan atau hingga subqueue dari semua worker sudah habis. Jika path masih belum ditemukan maka proses BFS akan masih berlanjut untuk depth berikutnya (ulangi langkah 3 dan 4).

3.2.2 Pemetaan Masalah menjadi Elemen Untuk Algoritma IDS

Elemen:

- Node Awal : Artikel awal Wikipedia.
- Node Tujuan : Artikel tujuan Wikipedia.
- Tujuan : Mencari jalur terpendek dari artikel awal ke artikel tujuan melalui link yang ada pada setiap artikel.
- Depth Limit: Batas kedalaman yang bertambah secara iteratif.
- Stack: Menyimpan node artikel yang perlu untuk diperiksa dan dieksplorasi lebih lanjut.
- Visited Map: Mencatat semua artikel yang sudah dikunjungi agar tidak terjadi pengulangan.
- Parent Map: Mencatat parent dari setiap node yang ada (hanya mencatat satu parent saja per artikel node)

Proses:

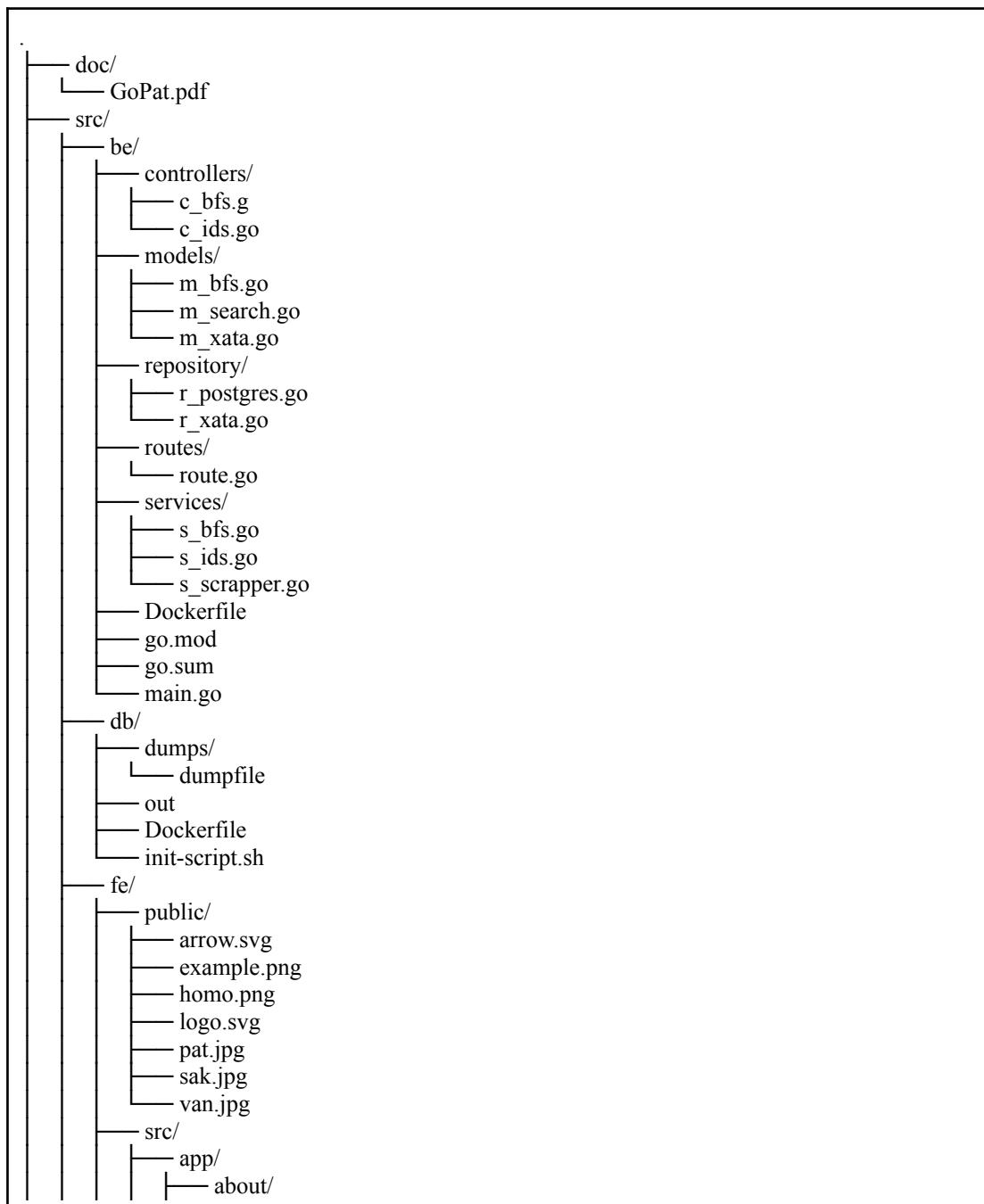
Penggunaan struktur data stack akan sedikit sulit jika ingin dilakukan paralelisme sehingga dalam tubes ini kode kami tidak menerapkan paralelisme pada IDS.

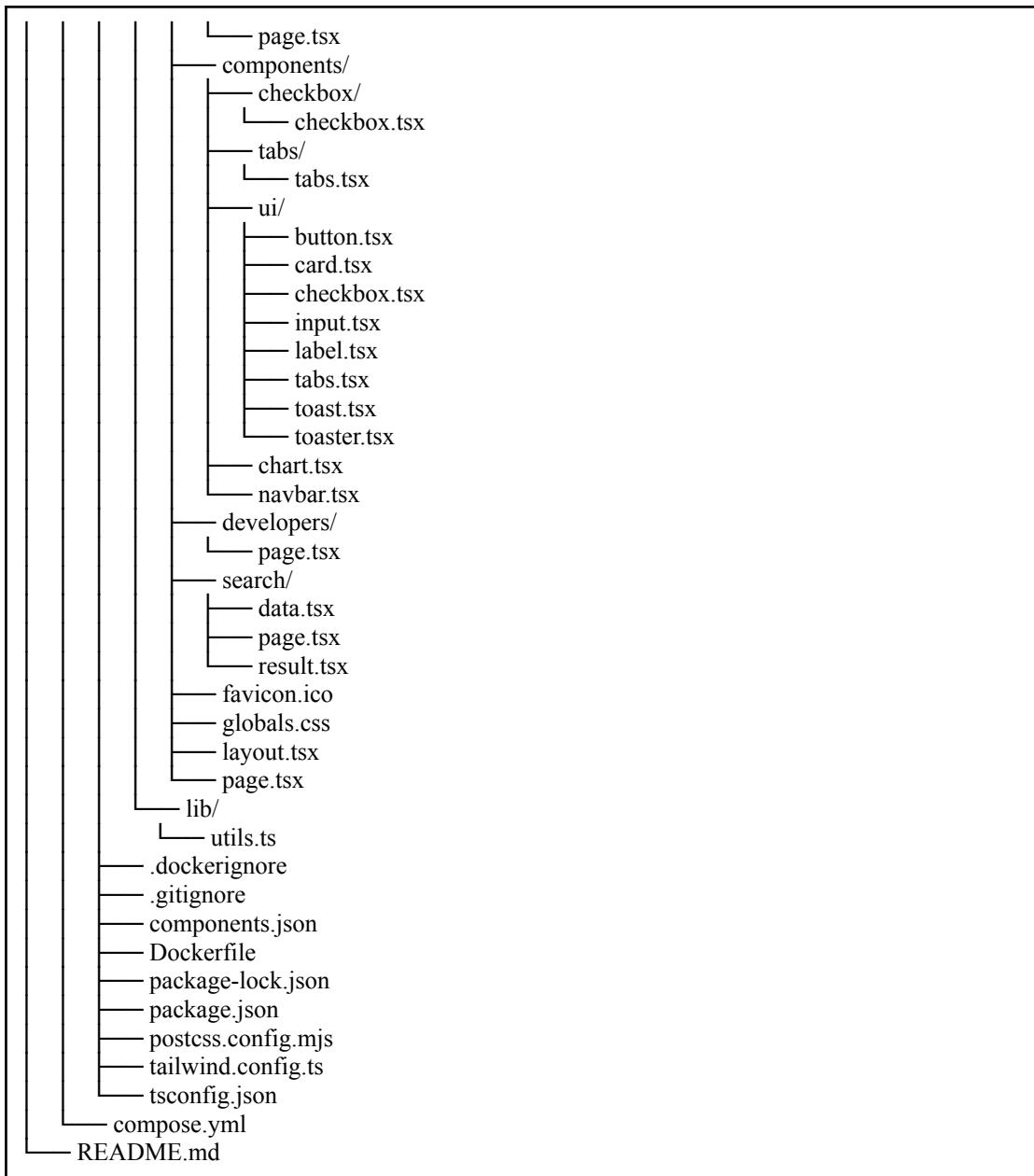
1. Batasan kedalam akan dimulai dari 0
2. Lakukan LDS untuk kedalaman yang sudah ditentukan
3. Dalam LDS akan terdapat stack dimana artikel awal akan dimasukkan pertama kedalam stack kemudian status visitednya akan diperbarui.
4. Link artikel yang berada pada paling atas stack akan diambil kemudian akan dicek apakah sama dengan artikel tujuan atau tidak jika sama maka path akan ditemukan dan jika tidak maka link tersebut akan dimasukkan kembali kedalam stack dan diubah status visitednya (link sebelumnya juga akan dicek apakah sudah dikunjungi atau belum). Data parent dari link tersebut juga akan ditambahkan ke dalam Parent Map. Jika ternyata suatu link tersebut sudah berada pada kedalaman maksimal maka link tersebut tidak akan dimasukkan kembali ke dalam stack dan data parent serta status visitednya juga tidak akan diubah.

5. Jika stack sudah kosong dan path masih belum ditemukan maka proses IDS akan berlanjut untuk kedalaman maksimal yang telah di *increment* satu dari sebelumnya.

3.3 Fitur Fungsional dan Arsitektur Aplikasi Web

Dalam pengembangan aplikasi web ini, kami memastikan struktur kode yang terorganisir dengan baik melalui pembagian dua direktori utama yaitu untuk frontend dan backend.





Aplikasi website ini dirancang dengan fokus pada pengalaman pengguna yang intuitif dan responsif. Fitur-fitur yang dihadirkan bertujuan untuk mengoptimalkan interaksi pengguna secara efektif dan efisien. Salah satunya adalah fitur pencarian yang mampu memberikan rekomendasi ketika pengguna mengetikkan sumber dan tujuan, disertai dengan tombol *swap* yang memudahkan pengguna untuk menukar kueri pencarian antara sumber dan tujuan dengan satu klik. Kemudian terdapat fitur untuk memilih algoritma yang dipilih dengan menggunakan component dari *shadcn* yang bernama *Tabs*. Terdapat juga tombol "Go" yang, saat diklik, akan memicu pengiriman permintaan *HTTP* ke backend server. Selanjutnya, respons dari server akan ditampilkan setelah diterima oleh aplikasi. Selain itu, kami juga menyediakan

grafik responsif untuk memvisualisasi data hasil pencarian rute terpendek dan memungkinkan pengguna untuk mendapatkan perspektif yang lebih jelas dari data yang disajikan. Fitur tambahan seperti halaman *About* dan *Dev* juga disertakan untuk memberikan konteks dan informasi lebih lanjut mengenai aplikasi dan tim di balik pengembangan aplikasi web ini.

Dalam segi arsitektur, di dalam root project terdapat folder *src* yang didalamnya terdapat dua folder yaitu frontend dan backend dan terdapat satu file yang bernama *compose.yml* yang berfungsi untuk menjalankan beberapa docker file secara langsung. Selain itu, terdapat juga folder *doc* yang isinya adalah laporan dari pembuatan tugas besar ini. Kemudian, terdapat *readme* file yang berisi tentang sedikit overview mengenai program ini dan juga tata cara memulai dan menggunakan aplikasi dan juga developer yang mengembangkan aplikasi web ini.

Dalam folder *fe*, program *frontend* mengorganisir file-file yang dibutuhkan untuk menjalankan aplikasi dalam subfolder *src/app*. Folder *public/* menyimpan asset statis seperti gambar yang digunakan dalam aplikasi frontend. Untuk meningkatkan modularitas, folder-folder ini dibagi lagi menjadi subfolder yang lebih spesifik. Sebagai contoh, dalam folder *'components'*, terdapat berbagai file seperti *'navbar'*, *'chart'*, dan komponen lain seperti *'tabs'* yang diimpor dari *shadcn*. Komponen-komponen ini kemudian digunakan dalam folder *'search'*, yang berisi program utama tempat proses pencarian dilaksanakan, yang terletak di file *'page.tsx'*.

Folder *'about'* menyajikan gambaran umum tentang aplikasi ini, termasuk penjelasan sedikit mengenai algoritma pencarian seperti BFS dan IDS yang digunakan. Sementara itu, folder *'developers'* menyediakan informasi tentang pengembang web.

Ada juga file seperti *'globals.css'*, *'page.tsx'*, dan *'layout.tsx'* yang bertindak sebagai file utama untuk mengatur gaya global dan komponen aplikasi utama yang digunakan untuk memanggil semua komponen yang telah dibuat seperti *'search'*, serta entry point untuk Next.js dalam aplikasi frontend. Terdapat pula *'tailwind.config.ts'* yang berfungsi sebagai file konfigurasi untuk Tailwind CSS.

Backend menggunakan bahasa golang dengan framework gin gonic. Dalam backend dibagi menjadi beberapa folder ada controllers, services, models, dan routes. Folder routes hanya terdiri dari satu file yang fungsinya untuk melakukan inisiasi

route atau endpoint yang nantinya akan dipanggil oleh frontend. Folder controllers berisi file-file yang mengandung controller yang berupa fungsi-fungsi yang nantinya akan dipanggil oleh route. Controller berfungsi untuk mengekstrak data dari body HTTP request dan juga mengekstrak data hasil proses backend menjadi HTTP Response yang sesuai.

Data request yang sudah diekstrak oleh controller nantinya akan dikirim ke services untuk diolah. Folder services berisi file-file yang mengandung algoritma utama dari program kami. Dalam services tertampung algoritma BFS dan IDS yang kami implementasikan beserta dengan algoritma scrapping yang kami gunakan. Secara umum services berisi logika utama dari program yang kami buat. Dalam struktur be juga terdapat folder repository yang berisi kode-kode yang berhubungan dengan database seperti query ke database dan lain sebagainya. Dalam tubes ini kami menggunakan database lokal berupa postgres sehingga pada folder repository terdapat file yang berisi query SQL ke database postgres tersebut.

Selain itu, ada juga folder models yang berisi file-file yang mengandung skema dari struktur data yang kami gunakan termasuk struktur dari body request. File main.go ditempatkan di root folder dari backend kami. Pada file main akan diinisialisasi gin dan kemudian akan dijalankan di port tertentu yang sudah kami tentukan.

Dikarenakan data caching kami simpan kedalam database dalam tugas besar ini terdapat folder tambahan yaitu db yang digunakan untuk mensetup database. Di dalam folder ini terdapat file .sh yang digunakan untuk menjalankan file dump sql yang berisi data caching ke dalam database. Di dalam folder db ini juga terdapat folder dumps dimana folder dumps ini akan dijadikan tempat untuk meletakkan dump file sql. Selain folder dumps terdapat juga folder out. Folder out ini ter-mount dengan folder yang ada di dalam docker sehingga ketika melakukan dump database dari dalam docker ke dalam bentuk sql file hasil dump tersebut juga bisa terlihat dari luar docker yaitu tepatnya pada folder out.

3.4 Ilustrasi Kasus

Program berbasis website yang kami buat ini pada dasarnya adalah program yang digunakan untuk menyelesaikan permainan wiki racer. Dengan menggunakan algoritma BFS dan IDS akan dapat dicari jarak terpendek antara dua buah artikel pada

wikipedia yang dapat ditempuh melalui berbagai macam hyperlink yang terdapat pada suatu artikel wikipedia tersebut.

Dalam algoritma *Breadth-First Search* (BFS), artikel awal akan disimpan sebagai titik awal dalam sebuah antrian. Sebuah struktur data berupa map yang bernama "visited" diinisialisasi dengan menandai titik awal sebagai sudah dikunjungi agar tidak dijelajahi lagi nantinya. Semua artikel yang terhubung, atau hyperlink, ditambahkan ke dalam antrian. Proses ini berlanjut selama antrian belum kosong dan artikel tujuan belum ditemukan. Setiap artikel yang dijelajahi akan ditandai sebagai sudah dikunjungi dalam map visited untuk mencegah pengulangan.

Untuk algoritma Iterative Deepening Search (IDS), mirip dengan BFS, artikel awal juga ditetapkan sebagai titik awal namun dimasukkan ke dalam sebuah tumpukan. Seperti dalam BFS, titik awal ini juga ditandai sebagai sudah dikunjungi. IDS dilakukan dengan metode DLS yang dilakukan berulang kali dengan maksimal kedalaman yang terus ditingkatkan jika target belum ditemukan. Artikel diambil dari puncak tumpukan, dan semua hyperlink yang dapat dijelajahi ditambahkan ke dalam tumpukan. Proses ini berlangsung hingga tumpukan kosong atau target ditemukan. Setiap simpul yang sudah masuk ke dalam stack akan ditandai sebagai sudah dikunjungi, dan jika target belum ditemukan, proses backtracking otomatis terjadi ketika simpul dikeluarkan dari tumpukan. Ketika stack sudah kosong dan path masih belum dilakukan maka proses DLS akan dilakukan kembali dengan kedalaman maksimal yang sudah ditingkatkan.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi dalam *Pseudocode*

```
func isExcluded(link string) bool
```



```
1 func isExcluded(link string) bool {  
2     if link == "/wiki/Main_Page" {  
3         return true  
4     }  
5  
6     for _, ns := range excludedNamespaces2 {  
7         if regexp.MustCompile(`^` + regexp.QuoteMeta(ns)).MatchString(link) {  
8             return true  
9         }  
10    }  
11    return false  
12 }
```

```
func ScrapeMultipleWikipediaLinks(urls []string, cache *sync.Map) ([]string, error)
```



```
1 func ScrapeMultipleWikipediaLinks(urls []string, cache *sync.Map) ([]string, error) {
2     // Create a collector
3     c := colly.NewCollector(
4         colly.Async(true),
5     )
6
7     // Create a queue with 2 consumer threads
8     q, err := queue.New(
9         100, // Number of consumer threads
10        &queue.InMemoryQueueStorage{MaxSize: len(urls)}, // Use default queue storage
11    )
12    if err != nil {
13        return nil, err
14    }
15
16    c.OnHTML("a[href]", func(e *colly.HTMLElement) {
17        link := e.Attr("href")
18        // Check if the link matches the combined regex
19        if combinedRegex.MatchString(link) {
20            fullLink := "https://en.wikipedia.org" + link
21            urlKey := e.Request.URL.String()
22            // Safely append the link to the slice for the URL
23            cache.Store(urlKey, func(value interface{}) interface{} {
24                if value == nil {
25                    return []string{fullLink}
26                }
27                return append(value.([]string), fullLink)
28            })
29        }
30    })
31
32    // Handle errors
33    c.OnError(func(r *colly.Response, err error) {
34        fmt.Println("Request URL:", r.Request.URL, "failed with response:", r, "\nError:", err)
35    })
36
37    // Add all URLs to the queue
38    for _, url := range urls {
39        q.AddURL(url)
40    }
41
42    // Consume URLs
43    q.Run(c)
44
45    // Wait until threads are finished
46    c.Wait()
47    return []string{}, nil
48 }
```

```
func DecodePercentEncodedString(encodedString string) string
```



```
1 func DecodePercentEncodedString(encodedString string) string {
2     decodedString, err := url.QueryUnescape(encodedString)
3     if err != nil {
4         return encodedString // return the error if the decoding fails
5     }
6     return decodedString
7 }
```

```
func ScrapeWikipediaLinks(url string) ([]string, error)
```



```
1 func ScrapeWikipediaLinks(url string) ([]string, error) {
2     if exist, err := repository.GetChildrenByParent(url); exist != nil {
3         if err != nil {
4             return nil, err
5         }
6         return exist, nil
7     }
8
9     result := make([]string, 0)
10
11    c := colly.NewCollector(
12        colly.AllowedDomains("wikipedia.org", "en.wikipedia.org"),
13    )
14
15    c.UserAgent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
16
17    c.SetRequestTimeout(15 * time.Second)
18
19    c.Limit(&colly.LimitRule{
20        Parallelism: 1,
21    })
22
23    c.OnHTML("a[href]", func(e *colly.HTMLElement) {
24        if !isVisible(e) {
25            return
26        }
27
28        link := e.Attr("href")
29        if combinedRegex.MatchString(link) {
30            fullLink := "https://en.wikipedia.org" + link
31            if !isExcluded(link) {
32                result = append(result, fullLink)
33            }
34        }
35    })
36
37    var attempt int
38    maxAttempts := 1
39    c.OnError(func(r *colly.Response, err error) {
40        if r.StatusCode == 0 { // Network error, possibly a timeout
41            attempt++
42            if attempt < maxAttempts {
43                fmt.Println("Retrying:", r.Request.URL)
44                r.Request.Retry()
45            } else {
46                fmt.Println("Request failed after retries:", r.Request.URL, "\nError:", err)
47            }
48        }
49    })
50
51    c.OnRequest(func(r *colly.Request) {
52    })
53
54    c.Visit(url)
55
56    if len(result)> 0 {
57
58        err := repository.SaveArticleWithChildren(url, result)
59        if err != nil {
60            fmt.Println(err)
61        }
62    }
63
64    return result, nil
65 }
```

```
func CompareArrays(arr1, arr2 []string) bool
```



```
1 func CompareArrays(arr1, arr2 []string) bool {
2     if len(arr1) != len(arr2) {
3         return false
4     }
5
6     countMap := make(map[string]int)
7
8     for _, item := range arr1 {
9         countMap[item]++
10    }
11
12    for _, item := range arr2 {
13        if countMap[item] == 0 {
14            return false
15        }
16        countMap[item]--
17    }
18
19    for _, count := range countMap {
20        if count != 0 {
21            return false
22        }
23    }
24
25    return true
26 }
```

```
func isVisible(e *colly.HTMLElement) bool
```



```
1 func isVisible(e *colly.HTMLElement) bool {
2     class := e.Attr("class")
3     class = strings.ReplaceAll(class, " ", "")
4     if strings.Contains(class, "nowraplinks") {
5         return false
6     }
7
8     // Check parent elements for visibility
9     for parent := e.DOM.Parent(); parent.Length() != 0; parent = parent.Parent() {
10
11         parentClass, found := parent.Attr("class")
12         parentClass = strings.ReplaceAll(parentClass, " ", "")
13         if found && strings.Contains(parentClass, "nowraplinks") {
14             return false
15         }
16     }
17     return true
18 }
```

```
func helperMulti(urls *list.List, goal string, visited *sync.Map, sem chan struct{}, wg
*sync.WaitGroup, count *uint32) [][]string
```

```

1 func helperMulti(urls *list.List, goal string, visited *sync.Map, sem chan struct{}, wg *sync.WaitGroup, count *uint32) [][]string {
2     var mu sync.Mutex
3     var allPath [][]string
4
5     size := urls.Len()
6
7     for i := 0; i < size; i++ {
8         path := urls.Remove(urls.Front()).([]string)
9         last := path[len(path)-1]
10        sem <- struct{}{}
11        wg.Add(1)
12        go func(url string, goal string) {
13            defer wg.Done()
14            defer func() { <-sem }()
15
16            res, _ := ScrapeWikipediaLinks(url)
17            for _, u := range res {
18                if u == goal {
19                    newPath := make([]string, len(path))
20                    copy(newPath, path)
21                    newPath = append(newPath, u)
22                    mu.Lock()
23                    allPath = append(allPath, newPath)
24                    fmt.Println(allPath)
25                    mu.Unlock()
26
27                } else {
28
29                    if _, exist := visited.LoadOrStore(u, true); !exist {
30                        newPath := make([]string, len(path))
31                        copy(newPath, path)
32                        newPath = append(newPath, u)
33                        atomic.AddUint32(count, 1)
34                        mu.Lock()
35                        urls.PushBack(newPath)
36                        mu.Unlock()
37                    }
38                }
39
40            }
41            {last, goal}
42        }
43
44    wg.Wait()
45
46    if len(allPath) > 0 {
47        fmt.Println(len(allPath))
48        return allPath
49    } else {
50        return nil
51    }
52
53 }

```

```
func AsyncBFSMulti(start, goal string) ([][]string, int)
```



```
1 func AsyncBFSMulti(start, goal string) ([][]string, int) {
2     var visited sync.Map
3     semp := make(chan struct{}, maxConcurrency)
4     var wg sync.WaitGroup
5     var countChecked uint32 = 1
6
7     if start == goal {
8         return [][]string{{start}}, 1
9     }
10
11    queue := list.New()
12    queue.PushBack([]string{start})
13    visited.Store(start, true)
14
15    i := 0
16    for queue.Len() > 0 {
17        i++
18        fmt.Println("DEPTH: ", i)
19        r := helperMulti(queue, goal, &visited, semp, &wg, &countChecked)
20        if r != nil {
21            return r, int(countChecked)
22        }
23    }
24
25    return nil, int(countChecked)
26 }
```

```
func AsyncBFS(start, goal string) ([]string, int)
```



```
1 func AsyncBFS(start, goal string) ([]string, int) {
2     var parent sync.Map
3     var visited sync.Map
4     var wg sync.WaitGroup
5     var mutex sync.Mutex
6     var countChecked uint32
7
8     var resultPath []string
9     var found bool
10
11    queue := []string{start}
12    visited.Store(start, true)
13
14    if start == goal {
15        return []string{start}, 1
16    }
17
18    for len(queue) > 0 && !found {
19        local := queue
20        queue = []string{}
21
22        batchsize := (len(local) / maxConcurrency) + 1
23        fmt.Println(batchsize)
24
25        for i := 0; i < len(local); i += batchsize {
26            end := i + batchsize
27            if end > len(local) {
28                end = len(local)
29            }
30            wg.Add(1)
31            go func(links []string) {
32                defer wg.Done()
33
34                for _, url := range links {
35                    res, _ := ScrapeWikipediaLinks(url)
36                    for _, link := range res {
37                        mutex.Lock()
38                        if found {
39                            mutex.Unlock()
40                            return
41                        }
42                        mutex.Unlock()
```



```
1      if link == goal {
2          path := []string{goal}
3          for at := url; at != start; {
4              path = append([]string{at}, path...)
5              tes, _ := parent.Load(at)
6              at = tes.(string)
7          }
8          path = append([]string{start}, path...)
9
10         mutex.Lock()
11         if !found {
12             resultPath = path
13             found = true
14             fmt.Println(resultPath)
15         }
16         mutex.Unlock()
17         return
18     }
19
20     if _, exist := visited.LoadOrStore(link, true); !exist {
21         atomic.AddUint32(&countChecked, 1)
22
23         parent.Store(link, url)
24         mutex.Lock()
25         if !found {
26             queue = append(queue, link)
27         } else {
28             mutex.Unlock()
29
30             return
31         }
32         mutex.Unlock()
33     }
34 }
35 }
36
37     }(local[i:end])
38 }
39 wg.Wait()
40 }
41
42 if found {
43     return resultPath, int(countChecked)
44 }
45
46 return nil, int(countChecked)
47 }
```

```
func ldsMulti(start string, goal string, maxDepth int, cached *map[string][]string, count *uint32) ([][]string, error)
```



```
1 func ldsMulti(start string, goal string, maxDepth int, cached *map[string][]string, count *uint32) ([][]string, error) {
2     var stack [][]string
3     stack = append(stack, []string{start})
4
5     visited := make(map[string]bool)
6     visited[start] = true
7
8     var paths [][]string
9     foundDepth := -1
10
11    for len(stack) > 0 {
12        n := len(stack) - 1
13        path := stack[n]
14        stack = stack[:n]
15
16        lastNode := path[len(path)-1]
17        currentDepth := len(path) - 1
18
19        if lastNode == goal {
20            if foundDepth == -1 || currentDepth == foundDepth {
21                paths = append(paths, path)
22                foundDepth = currentDepth
23                continue
24            }
25        }
26
27        if foundDepth != -1 && currentDepth >= foundDepth {
28            continue
29        }
```



```
1
2     if currentDepth < maxDepth {
3         var links []string
4         var err error
5
6         if cachedLinks, ok := (*cached)[lastNode]; ok {
7             links = cachedLinks
8         } else {
9             links, err = ScrapeWikipediaLinks(lastNode)
10            if err != nil {
11                return nil, err
12            }
13            (*cached)[lastNode] = links
14            (*count)++
15        }
16
17        for _, link := range links {
18            if !visited[link] {
19                visited[link] = true
20                newPath := append([]string(nil), path...)
21                newPath = append(newPath, link)
22                stack = append(stack, newPath)
23            }
24        }
25    }
26}
27
28    if len(paths) > 0 {
29        return paths, nil
30    }
31    return nil, fmt.Errorf("path not found")
32 }
```

```
func lds(start string, goal string, maxDepth int, cached *map[string][]string, count *uint32) ([]string)
```

●

●

●

```
1 func lds(start string, goal string, maxDepth int, cached *map[string][]string, count *uint32) ([]string) {
2     var stack [][]string
3     stack = append(stack, []string{start}) // Using slice as stack
4
5     visited := make(map[string]bool)
6     visited[start] = true
7
8     for len(stack) > 0 {
9         // Pop from the stack
10        n := len(stack) - 1
11        path := stack[n]
12        stack = stack[:n]
13
14        lastNode := path[len(path)-1]
15        currentDepth := len(path) - 1
16
17        if lastNode == goal {
18            return path// Return the path immediately when the goal is found
19        }
20
21        if currentDepth < maxDepth {
22            var links []string
23            // var err error
```

```
1
2     if cachedLinks, ok := (*cached)[lastNode]; ok {
3         links = cachedLinks
4     } else {
5         links, _ = ScrapeWikipediaLinks(lastNode)
6
7         (*cached)[lastNode] = links
8     }
9
10    for _, link := range links {
11        if !visited[link] {
12            visited[link] = true
13            (*count)++ // Increment the counter for each scrape
14            newPath := append([]string(nil), path...) // Make a copy of the path
15            newPath = append(newPath, link)
16            stack = append(stack, newPath) // Push to the stack
17        }
18    }
19}
20}
21
22    return nil
23 }
```

```
func IDS(start string, goal string, maxDepth int) ([]string, int)
```



```
1 func IDS(start string, goal string, maxDepth int) ([]string, int){  
2     cached := make(map[string][]string)  
3  
4     if(start == goal){  
5         return []string{start}, 1  
6     }  
7     var countChecked uint32  
8     i := 0  
9     for {  
10         if (i >= maxDepth){  
11             break  
12         }  
13  
14         path := lds(start, goal, i, &cached, &countChecked)  
15         if path != nil {  
16             return path, int(countChecked)  
17         }  
18         i++  
19  
20     }  
21  
22     return nil, int(countChecked)  
23 }
```

4.2 Spesifikasi Teknis Program dan Struktur Data Program

4.2.1 Algoritma BFS

Dalam implementasi algoritma Breadth-First Search (BFS) untuk mencari jalur antara dua artikel Wikipedia, antrian (queue) digunakan sebagai struktur data kunci untuk mengelola urutan eksplorasi node atau simpul. Proses ini diawali dengan menambahkan artikel sumber ke dalam antrian dan menandainya sebagai telah dikunjungi, untuk menghindari revisi dan duplikasi dalam penjelajahan. Setelah artikel awal dimasukkan, algoritma memulai loop utama yang berlangsung selama masih ada elemen dalam antrian. Dalam setiap iterasi, artikel yang berada di depan antrian diambil dan dijadikan fokus

eksplorasi saat itu. Selama proses ini, algoritma mencari semua hyperlink yang tersedia pada artikel tersebut. Jika tautan mengarah pada artikel tujuan, maka jalur telah ditemukan. Jika tidak, Setiap tautan yang ditemukan dan belum dikunjungi akan ditambahkan ke dalam antrian dan siap untuk dieksplorasi pada iterasi berikutnya.

Langkah-langkah ini memastikan bahwa eksplorasi berlangsung secara melebar (breadth-first), mengutamakan artikel yang lebih dekat dengan sumber sebelum beranjak ke kedalaman yang lebih jauh. Pendekatan ini sangat efektif untuk menjelajahi jaringan yang luas dan terhubung, seperti halnya pada Wikipedia, di mana setiap artikel potensial bisa memiliki sejumlah tautan yang mengarah ke halaman lain. Antrian membantu dalam mengatur dan memprioritaskan artikel-artikel ini berdasarkan urutan mereka ditambahkan, sehingga memungkinkan algoritma untuk secara sistematis memproses dan menelusuri setiap kemungkinan rute dari artikel sumber menuju ke tujuan. Dengan demikian, BFS memungkinkan penemuan jalur terpendek atau jalur potensial antara dua halaman dengan cara yang terorganisir dan efisien, memaksimalkan penggunaan informasi yang tersedia sambil meminimalkan upaya yang tidak perlu.

Aspek yang unik dari implementasi ini adalah penggunaan paralelisme. Dengan memanfaatkan goroutines, sebuah fitur dalam Go yang memungkinkan eksekusi tugas secara konkuren, proses eksplorasi dapat dipercepat secara signifikan. Paralelisme diatur dengan batasan yang ditentukan untuk mengontrol jumlah tugas yang berjalan bersamaan, memungkinkan beberapa artikel diproses secara simultan. Ini sangat berguna dalam jaringan besar seperti Wikipedia, di mana jumlah tautan dan potensi jalur yang harus dieksplorasi bisa sangat besar. Dalam kasus BFS ini paralelisme dilakukan dengan cara membagi queue menjadi sejumlah bagian sesuai dengan jumlah go routine atau worker yang dibuat. Pembagian queue ini dilakukan pada setiap kedalaman untuk memastikan proses yang berlangsung secara paralel berada pada kedalaman yang sama sehingga jalur yang ditemukan selalu berada pada kedalaman yang sesuai dan merupakan yang terpendek.

Menggunakan goroutines tidak hanya meningkatkan efisiensi waktu dalam menemukan jalur tetapi juga memanfaatkan sumber daya komputasi

secara lebih optimal. Penggunaan mutex dan `sync.Map` juga krusial dalam mengelola akses ke sumber daya bersama, seperti antrian dan status kunjungan, untuk mencegah konflik dan kesalahan data. Dengan cara ini, BFS yang dilakukan tidak hanya sistematis dalam menjelajahi semua artikel yang mungkin tetapi juga sangat efisien, memastikan bahwa setiap jalur yang mungkin ditemukan dengan cepat dan akurat.

4.2.2 Algoritma IDS

Dalam algoritma Iterative Deepening Search (IDS), penggunaan struktur data tumpukan (*stack*) sangat krusial untuk mengatur proses pencarian yang serupa dengan *Depth-First Search* (DFS). Pada IDS, DFS diimplementasikan berulang dengan meningkatkan batas kedalaman pada setiap iterasinya. DFS dengan batas kedalaman tersebut biasa disebut juga DLS (*Deep Limiting Search*). Proses dimulai dari simpul awal, dengan tumpukan yang berperan penting dalam menyimpan simpul-simpul selama pencarian mencapai kedalaman yang lebih jauh dalam setiap siklus. Jika batas kedalaman yang ditetapkan telah tercapai untuk semua node dan solusi belum ditemukan, IDS akan memulai kembali pencarian dari simpul awal dan meningkatkan batas kedalaman untuk iterasi berikutnya. Penerapan *stack* memungkinkan IDS untuk menyelami lebih dalam pada graf, mengulang dari awal dan secara bertahap memperluas batas kedalaman. Hal ini menghasilkan pencarian yang sistematis dan efisien, memanfaatkan memori yang relatif kecil untuk menghadapi ruang pencarian besar atau kompleks. Setiap tingkat kedalaman dieksplorasi sepenuhnya sebelum beranjak ke tingkat yang lebih tinggi, yang memastikan pendekatan yang terkontrol dalam menemukan solusi. Ini membantu kelemahan DFS yang bisa terjebak dalam cabang yang dalam dan tidak efisien dalam menentukan solusi terdekat jika tersembunyi di belakang simpul yang yang lebih dangkal.

Dalam penentuan rute antara dua artikel Wikipedia, tumpukan dalam algoritma IDS memainkan peran kunci dalam menjaga efisiensi proses penjelajahan. IDS dimulai pada kedalaman yang rendah dan meningkat secara bertahap, dengan tumpukan yang mengatur urutan eksplorasi simpul dalam graf yang mewakili hubungan antar artikel. Artikel yang dijadikan simpul awal dimasukkan ke dalam tumpukan, dan IDS berusaha menelusuri dari

artikel tersebut dengan batasan kedalaman tertentu. Jika batas kedalaman tercapai tanpa menemukan artikel tujuan, IDS akan restart dengan kedalaman yang lebih besar. Dalam setiap siklus, simpul yang dieksplorasi pada kedalaman tertentu disimpan dalam tumpukan dan dieksplorasi satu per satu. Setiap kali simpul terhubung dengan artikel lain, koneksi tersebut ditambahkan ke tumpukan untuk dieksplorasi, asalkan masih dalam batas kedalaman yang diizinkan. Dengan cara ini, IDS memanfaatkan tumpukan untuk secara bertahap dan strategis mendalam ke dalam graf artikel Wikipedia, memastikan pencarian menyeluruh terhadap semua rute potensial antara dua artikel.

Penerapan DLS dalam IDS ini sebenarnya bisa juga dengan cara menggunakan rekursif namun kami menghindarinya dan lebih memilih menggunakan stack. Hal tersebut kami lakukan dikarenakan penggunaan rekursif memberikan kemungkinan adanya overhead di setiap pemanggilan fungsi rekursif.

Dikarenakan penggunaan struktur data stack pada IDS penerapan paralelisme menjadi lebih sulit untuk dilakukan dikarenakan sulitnya menentukan urutan node yang tepat untuk dieksplorasi. Oleh sebab itu dalam kode yang kami buat kami tidak menerapkan paralelisme. Penerapan paralelisme dalam IDS membuat algoritma IDS

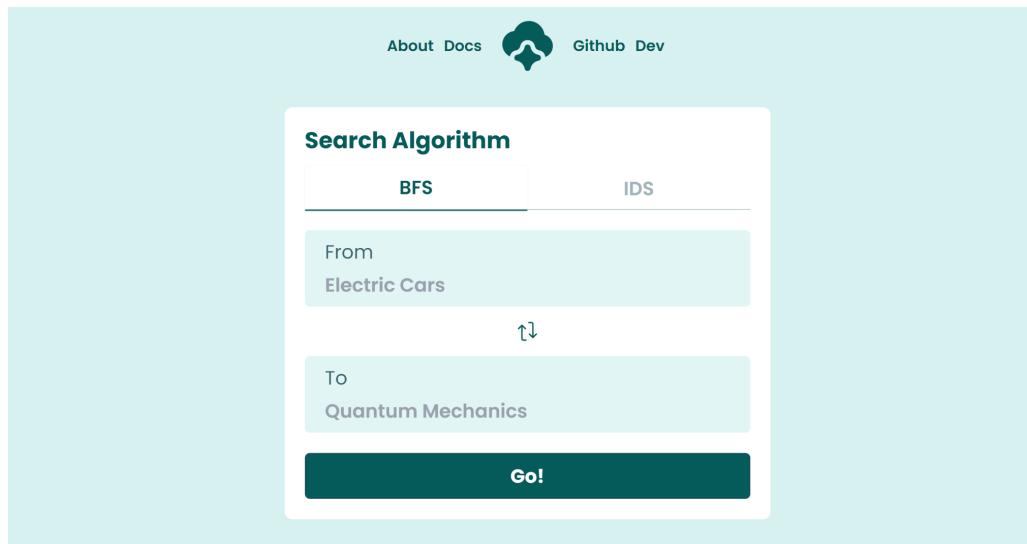
4.3 Tata Cara Penggunaan Program

Program harus dijalankan pada sistem yang sudah meng-install *Node.js* dan bahasa pemrograman *Go* dan juga sudah memiliki *Docker*. Berikut adalah langkah-langkah untuk memulai program WikiRace by GoPat:

1. Lakukan *git clone* dari repositori program ini dengan mengetik *git clone https://github.com/Otzzu/Tubes2_GoPat* pada terminal.
2. Masuk ke dalam root dari program kami kemudian masuk ke dalam folder src dengan menuliskan *cd src*.
3. Ketika sudah berada dalam folder src ketikan *docker compose up -build* lalu tunggu hingga semua proses (kontainer docker) sudah berjalan (tidak menggunakan data caching).
4. Jika sudah dipastikan bahwa semua proses sudah berjalan semua dengan baik buka browser dan akses <http://localhost:3000>

5. Data caching berupa dump sql dapat didownload pada link berikut:
<https://drive.google.com/drive/folders/1TkPVTfJu7VWU9aeWAx-jk1esvYSHhE0p?usp=sharing>
Ukuran dump file yang paling besar berisi data cache paling banyak
6. Jika ingin menggunakan data caching pastikan untuk menghapus semua kontainer dan volume pada docker untuk proyek ini sehingga projek bisa diinisialisasi ulang dengan baik. Jika mau lebih aman bisa juga menghapus image yang sudah ada sebelumnya. Proses penghapusan ini bisa dilakukan melalui docker desktop. Setiap kali ingin melakukan load dump.sql baru selalu lakukan proses penghapusan ini atau lakukan load dump manual melakukan bash di dalam docker secara manual.
7. Untuk menggunakan data caching pindahkan data dump sql ke dalam folder /src/db/dumps. Jika belum ada folder dumps buat saja dan pastikan nama file dump adalah *dumpfile.sql*.
8. Pada terminal, kembali ke folder src dan jalankan *docker compose up -build* dan tunggu hingga proses sudah berjalan semua dengan baik (dikarenakan ukuran dump sangat besar, proses ini mungkin akan berjalan cukup lama dan kontainer untuk *backend* akan te-restart terus-terusan hingga data dari database siap dan backend bisa terhubung dengan baik dengan database).

Berikut adalah tampilan program web setelah dijalankan:



Berikut adalah tata cara penggunaan program WikiRace:

1. Pengguna dapat memilih algoritma yang ingin digunakan, yaitu BFS atau IDS.

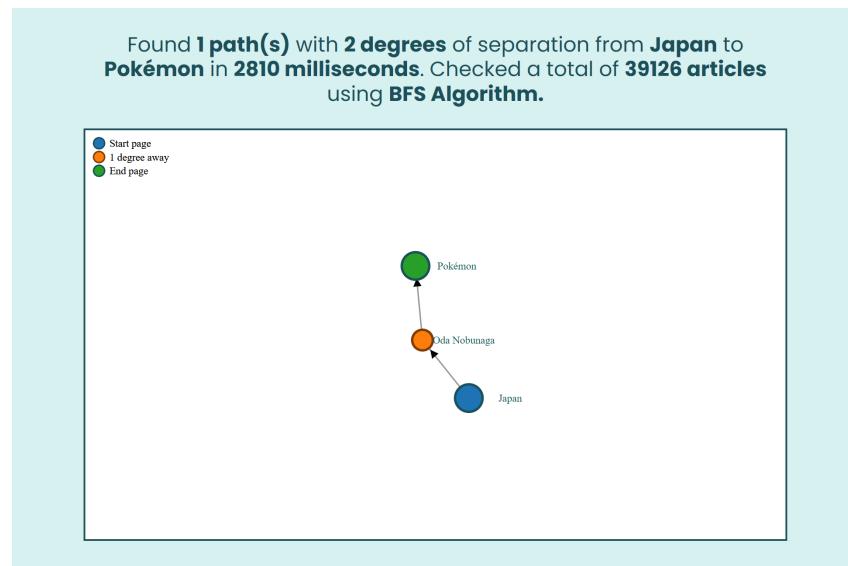
2. Pengguna harus memasukkan judul artikel awal dan akhir, lalu menekan hasil rekomendasinya.

Search Algorithm	
BFS	IDS
From japan	
<ul style="list-style-type: none">JapanJapanese languageJapan Air Lines Flight 123Japanese war crimesJapanese invasions of Korea (1592–1598)Japan Self-Defense ForcesJapanese yen	
	From Japan
	<p>↑↓</p>
	To Pokémon
	Go!

3. Pengguna juga dapat menggunakan tombol swap untuk menukar posisi input awal dan tujuan jika diperlukan.

Search Algorithm	
BFS	IDS
From Pokémon	
<p>↑↓</p>	
To Japan	
	Go!

4. Klik tombol GO! untuk menampilkan hasil pencarian dan visualisasi chartnya.



4.4 Hasil Pengujian

1. Test Case 1 (2 degrees)

a. BFS *Single Path*

Search Algorithm

BFS	IDS
------------	------------

MultiPath

From
Jokowi

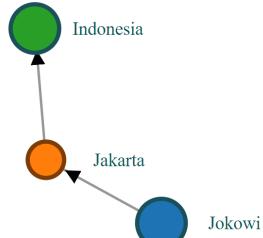
To
Indonesia

Go!

Found **1 path(s)** with **2 degrees** of separation from **Jokowi** to **Indonesia** in **17 milliseconds**. Checked a total of **359 articles** using **BFS Algorithm**.

Found **1 path(s)** with **2 degrees** of separation from **Jokowi** to **Indonesia** in **17 milliseconds**. Checked a total of **359 articles** using **BFS Algorithm**.

Start page
1 degree away
End page



b. BFS Multi Path

Search Algorithm

BFS

IDS

MultiPath

From

Jokowi



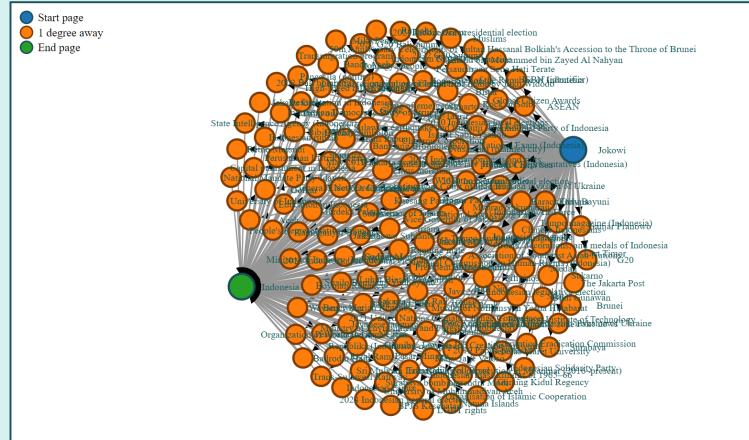
To

Indonesia

Go!

Found **167 path(s)** with **2 degrees** of separation from **Jokowi** to **Indonesia** in **3151 milliseconds**. Checked a total of **41726 articles** using **BFS Algorithm**.

Found **167 path(s)** with **2 degrees** of separation from **Jokowi** to **Indonesia** in **3151 milliseconds**. Checked a total of **41726 articles** using **BFS Algorithm**.



c. IDS

Search Algorithm

BFS **IDS**

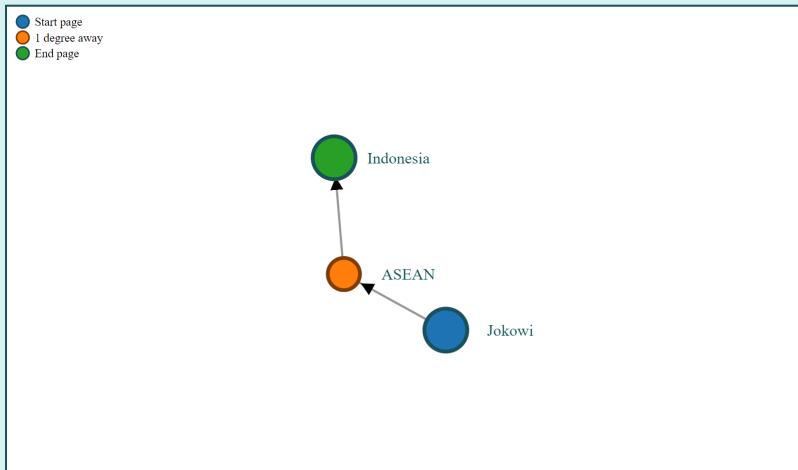
From
Jokowi

To
Indonesia

Go!

Found **1 path(s)** with **2 degrees** of separation from **Jokowi** to **Indonesia** in **2 milliseconds**. Checked a total of **1056 articles** using **IDS Algorithm**.

Found **1 path(s)** with **2 degrees** of separation from **Jokowi** to **Indonesia** in **2 milliseconds**. Checked a total of **1056 articles** using **IDS Algorithm**.



2. Test Case 2 (3 degrees)

a. BFS Single Path

Search Algorithm

BFS **IDS**

MultiPath

From
China

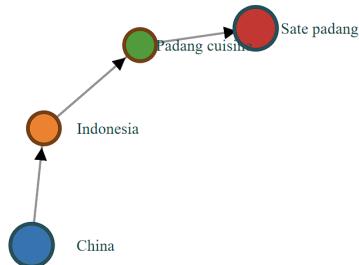
To
Sate padang

Go!

Found **1 path(s)** with **3 degrees** of separation from **China** to **Sate padang** in **9604 milliseconds**. Checked a total of **332060 articles** using **BFS Algorithm**.

Found **1 path(s)** with **3 degrees** of separation from **China** to **Sate padang** in **9604 milliseconds**. Checked a total of **332060 articles** using **BFS Algorithm**.

- Start page
- 1 degree away
- 2 degrees away
- End page



b. BFS Multi Path

Search Algorithm

BFS

IDS

MultiPath

From
China

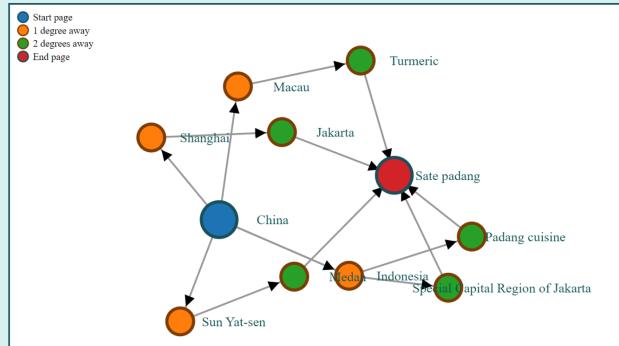


To
Sate padang

Go!

Found **5 path(s)** with **3 degrees** of separation from **China** to **Sate padang** in **363318 milliseconds**. Checked a total of **2391998 articles** using **BFS Algorithm**.

Found **5 path(s)** with **3 degrees** of separation from **China** to **Sate padang** in **363318 milliseconds**. Checked a total of **2391998 articles** using **BFS Algorithm**.



c. IDS

Search Algorithm

BFS

IDS

From

China



To

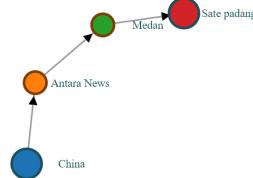
Sate padang

Go!

Found **1 path(s)** with **3 degrees** of separation from **China** to **Sate padang** in **52638 milliseconds**. Checked a total of **182875 articles** using **IDS Algorithm**.

Found **1 path(s)** with **3 degrees** of separation from **China** to **Sate padang** in **52638 milliseconds**. Checked a total of **182875 articles** using **IDS Algorithm**.

Start page
1 degree away
2 degrees away
End page



3. Test Case 3 (3 degrees)

a. BFS Single Path

Search Algorithm

BFS

IDS

MultiPath

From

Japan

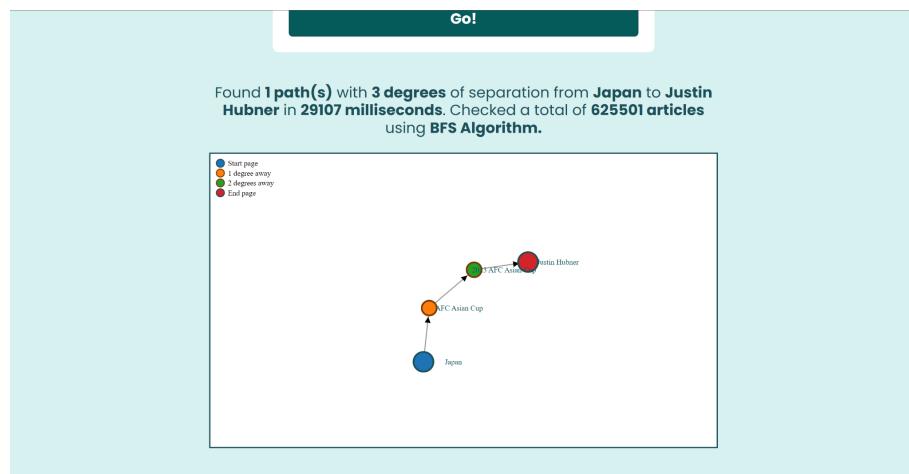


To

Justin Hubner

Go!

Found **1 path(s)** with **3 degrees** of separation from **Japan** to **Justin Hubner** in **29107 milliseconds**. Checked a total of **625501 articles** using **BFS Algorithm**.

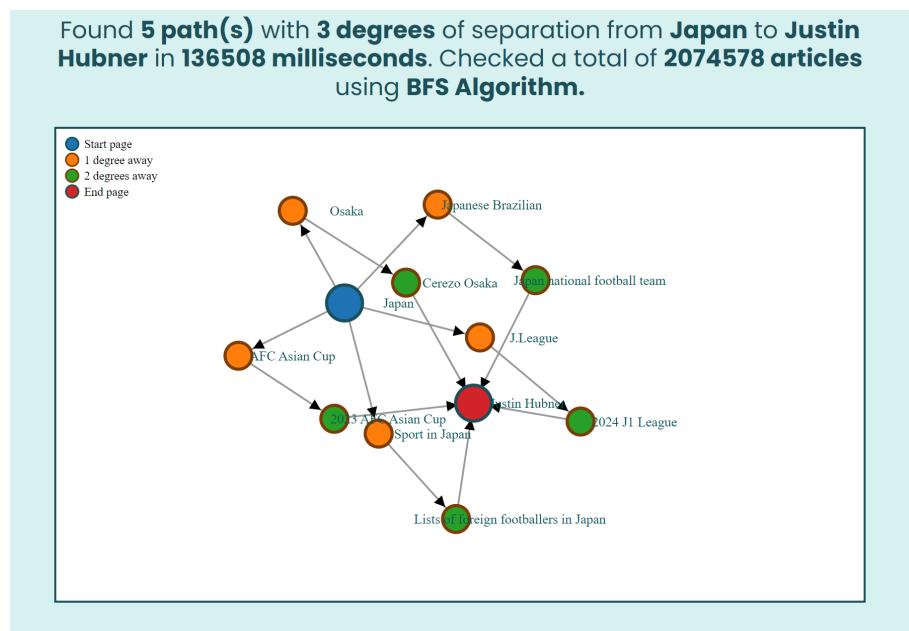


b. BFS Multi Path

Search Algorithm

BFS	IDS
<input checked="" type="checkbox"/> MultiPath	
From Japan	
↑↓	
To Justin Hubner	
Go!	

Found **5 path(s)** with **3 degrees** of separation from **Japan** to **Justin Hubner** in **136508 milliseconds**. Checked a total of **2074578 articles** using **BFS Algorithm**.



c. IDS

Search Algorithm

BFS **IDS**

From
Japan

To
Justin Hubner

Go!

Found 1 path(s) with 3 degrees of separation from **Japan** to **Justin Hubner** in 1146322 milliseconds. Checked a total of 407685 articles using **IDS Algorithm**.

Start page

Found 1 path(s) with 3 degrees of separation from **Japan** to **Justin Hubner** in 1146322 milliseconds. Checked a total of 407685 articles using **IDS Algorithm**.

Start page
1 degree away
2 degrees away
Total page

```
graph TD; Japan((Japan)) --> JLeague((J League)); JLeague --> JustinHubner((Justin Hubner))
```

4. Test Case 4 (4 degrees)

a. BFS Single Path

Search Algorithm

BFS
IDS

MultiPath

From

To

Go!

Found **1 path(s)** with **4 degrees** of separation from **Aglet** to **Sate padang** in **35134 milliseconds**. Checked a total of **904133 articles** using **BFS Algorithm**.

Found **1 path(s)** with **4 degrees** of separation from **Aglet** to **Sate padang** in **35134 milliseconds**. Checked a total of **904133 articles** using **BFS Algorithm**.

- Start page
- 1 degree away
- 2 degrees away
- 3 degrees away
- End page

b. BFS *Multi Path*

Search Algorithm

BFS	IDS
<input checked="" type="checkbox"/> MultiPath	
From Aglet	
↑↓	
To Sate padang	
Go!	

Found **5 path(s)** with **4 degrees** of separation from **Aglet** to **Sate padang** in **267752 milliseconds**. Checked a total of **3135074 articles** using **BFS Algorithm**.

Found **5 path(s)** with **4 degrees** of separation from **Aglet** to **Sate padang** in **267752 milliseconds**. Checked a total of **3135074 articles** using **BFS Algorithm**.

```

graph TD
    Aglet((Aglet)) --> ManilaHemp((Manila hemp))
    ManilaHemp --> Turmeric((Turmeric))
    Turmeric --> SatePadang((Sate padang))
    SatePadang --> PadangCuisine((Padang cuisine))
    PadangCuisine --> Jakarta((Jakarta))
    Jakarta --> Medan((Medan))
    Medan --> SpecialJavaneseRegion((Special Javanese Region of Jakarta))
    SpecialJavaneseRegion --> Indonesia((Indonesia))
    Indonesia --> Aglet
    Indonesia --> SatePadang
    
```

c. IDS

Search Algorithm

BFS	IDS
From Aglet	
To Sate padang	
Go!	

Found **1 path(s)** with **4 degrees** of separation from **Aglet** to **Sate padang** in **3517 milliseconds**. Checked a total of **404096 articles** using **IDS Algorithm**.

Found **1 path(s)** with **4 degrees** of separation from **Aglet** to **Sate padang** in **3517 milliseconds**. Checked a total of **404096 articles** using **IDS Algorithm**.

● Start page
● 1 degree away
● 2 degrees away
● 3 degrees away
● End page

4.5 Analisis Pengujian

Secara umum algoritma BFS seharusnya bisa lebih cepat dikarenakan terdapat proses paralelisme pada BFS dibandingkan dengan IDS. Namun jika semua data sudah di cache maka proses IDS terkadang akan bisa lebih cepat karena tidak perlu melakukan scraping. Namun jika harus dilakukan bersamaan dengan scraping maka kemungkinan paling besar BFS akan lebih cepat. Proses IDS terkadang bisa lebih cepat jika artikel yang dicari berada

dekat dengan atas stack. Namun jika ternyata artikel yang dicari berada jauh di dasar sebuah stack maka pencarian IDS pasti akan lama. Hal ini juga berlaku pada BFS semakin awal posisi artikel tujuan pada queue maka waktu eksekusi BFS akan bisa lebih cepat. Penggunaan paralelisme secara umum akan memakan cukup banyak memori sehingga akan diperlukan waktu lebih untuk mematikan program karena terdapat beberapa worker (*go routine*) yang perlu dimatikan. Dalam BFS terkadang jalur dari awal ke tujuan sudah ditemukan namun backend api belum mengembalikan response pada frontend karena terdapat waktu lebih yang diperlukan untuk mematikan worker-worker yang ada. Dalam IDS dikarenakan tidak ada paralelisme sama sekali maka ketika jalur sudah ditemukan, backend api bisa langsung memberikan *response* kepada frontend dan hasil bisa segera ditampilkan. Dalam test case yang dilakukan di atas IDS cenderung lebih cepat dikarenakan pengetesan selalu dilakukan dengan BFS terlebih dahulu sehingga data sudah di cache terlebih dahulu dan IDS bisa berjalan lebih cepat.

Dalam tubes ini kami menggunakan 10 worker untuk menjalankan paralelisme dalam algoritma BFS kami. Jumlah worker ini sudah merupakan jumlah yang paling tepat dikarenakan jika jumlahnya sudah terlalu banyak maka akan lebih mudah terkena error to many request dari wikipedia. Jumlah worker di bawah 10 juga kurang efektif karena proses pencarian yang dilakukan menjadi terlalu lama. Dalam caching, data akan selalu diambil dari database dan terkadang proses pengambilan data dari database ini juga memakan waktu jika terlalu banyak query yang dikirimkan ke database terutama pada proses paralel di BFS.

Paralelisme sebenarnya juga tidak selalu memberikan hasil yang terbaik dikarenakan terdapat terlalu banyak data yang perlu di-share di antara para worker yang mengakibatkan terlalu banyaknya proses lock dan unlock. Proses lock dan unlock dengan menggunakan mutex ini terkadang juga menghambat proses asinkron itu sendiri.

BAB V

PENUTUP

5.1 Kesimpulan

Melalui tugas besar ini telah dibuatkan sebuah aplikasi berbasis web yang digunakan untuk mencari jalur terpendek di antara dua buah artikel pada wikipedia melalui hyperlink-hyperlink yang ada pada setiap artikel tersebut. Aplikasi ini dibuat dengan menggunakan next.js untuk frontend dan golang untuk backend lalu digunakan juga postgreSQL sebagai databasenya. Aplikasi ini juga dibungkus ke dalam aplikasi docker sehingga lebih mudah dalam menjalankannya.

Berdasarkan analisis yang sudah kami lakukan kami menemukan bahwa proses scrapping benar-benar memakan waktu. Hal itu terlihat dari hasil waktu eksekusi yang bisa lebih cepat ketika proses caching sudah diterapkan menggunakan database postgreSQL. Penerapan database ini dipilih karena proses query yang cukup cepat dan agar data caching bisa persisten secara lebih lama.

Algoritma BFS yang kami terapkan menggunakan proses paralelisme dengan memanfaatkan fitur go routine yang telah disediakan oleh bahasa golang. Dengan penerapan proses paralelisme ini algoritma BFS ini bisa berjalan dengan lebih cepat dan efisien. Untuk algoritma IDS sendiri kami tidak menerapkan proses paralelisme dan hanya mengandalkan proses *synchronous* dan iterasi biasa. Dalam kasus biasa tanpa adanya caching algoritma IDS akan cenderung lebih lama dikarenakan tidak adanya proses paralelisme dan ada banyaknya proses pengunjungan artikel berulang yang cukup memakan waktu. Penerapan algoritma IDS sudah kami buat seefisien mungkin dengan menerapkan stack daripada menggunakan pemanggilan fungsi rekursif untuk menghindari adanya overhead waktu pemanggilan.

Dalam beberapa kasus algoritma IDS bisa lebih cepat daripada algoritma BFS dikarenakan dalam algoritma BFS diperlukan waktu lebih untuk mematikan semua worker yang telah dibangkitkan dalam proses BFS tersebut. Proses mematikan worker tersebut terkadang memakan waktu yang cukup lama yang membuat proses pengembalian hasil dalam fungsi BFS menjadi tertunda.

Secara keseluruhan algoritma BFS dan IDS ini telah berhasil dalam menyelesaikan permasalah pencarian rute terpendek diantara dua buah artikel wikipedia terutama untuk kedalaman atau degree di bawah lima.

5.2 Saran

Untuk meningkatkan efisiensi pencarian artikel menggunakan metode BFS dan IDS, kami mengusulkan penggunaan database yang telah terisi data Wikipedia yang telah diolah sebelumnya. Proses preprocessing ini bertujuan untuk meminimalkan beban scraping selama pencarian berlangsung, sehingga semua proses pencarian dapat berlangsung dengan lebih lancar.

Kami juga menyarankan penerapan paralelisme dalam IDS untuk menambah kecepatan. Meskipun fitur ini belum terintegrasi dalam program saat ini, penerapan paralelisme bisa meningkatkan kecepatan pencarian secara signifikan dengan memungkinkan pencarian yang berlangsung secara simultan di berbagai cabang. Kami masih belum menerapkan parallelism pada IDS dikarenakan kesulitan dalam mensinkronisasi berbagai macam worker yang bekerja bersamaan

Untuk mempercepat BFS, kami merekomendasikan penggunaan teknik double-ended BFS. Metode ini memulai BFS dari artikel sumber dan tujuan secara bersamaan. Dengan penggunaan double-ended BFS proses pencarian akan jauh menjadi lebih efektif. Double-ended BFS disini berlangsung secara heuristic dimana pada mulanya akan diasumsikan bahwa *path* dari tujuan ke titik temu merupakan *path* dua arah. Ketika proses pencarian dari sumber dan tujuan bertemu di tengah maka akan dilakukan pengecekan kembali untuk menentukan apakah *path* tersebut valid atau tidak. Ketika *path* memang benar valid dari sumber ke tujuan maka baru bisa dikatakan bahwa hasil telah ditemukan jika ternyata tidak valid maka pencarian akan kembali dilanjutkan.

Proses scraping wikipedia bukanlah hal yang mudah karena terdapat berbagai macam batasan seperti limit request. Mengirimkan request yang berlebihan akan mengakibatkan terjadinya pemblokiran dari Wikipedia oleh sebab itu seharusnya untuk mengatasi hal tersebut bisa dengan memanfaatkan VPN, DNS, atau proxy.

Dalam wikipedia juga sebenarnya terdapat beberapa link yang di hidden oleh sebab itu sebaiknya proses scraping bisa dibuat lebih baik lagi sehingga hanya mendapatkan link-link yang valid dan tampak terlihat di tampilan websitenya.

DAFTAR PUSTAKA

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

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/matdis23-24.htm>

LAMPIRAN

Pranala Repository :

https://github.com/Otzzu/Tubes2_GoPat

Pranala video YouTube :

Pranala link dumpsql :

<https://drive.google.com/drive/folders/1TkPVTfJu7VWU9aeWAx-jk1esvYSHhE0p>