LAPORAN TUGAS BESAR 2

IF2211 STRATEGI ALGORITMA

Disusun oleh:

Muhammad Naufal Aulia	(13522074)
Keanu Amadius Gonza Wrahatno	(13522082)
Naufal Adnan	(13522116)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

2024

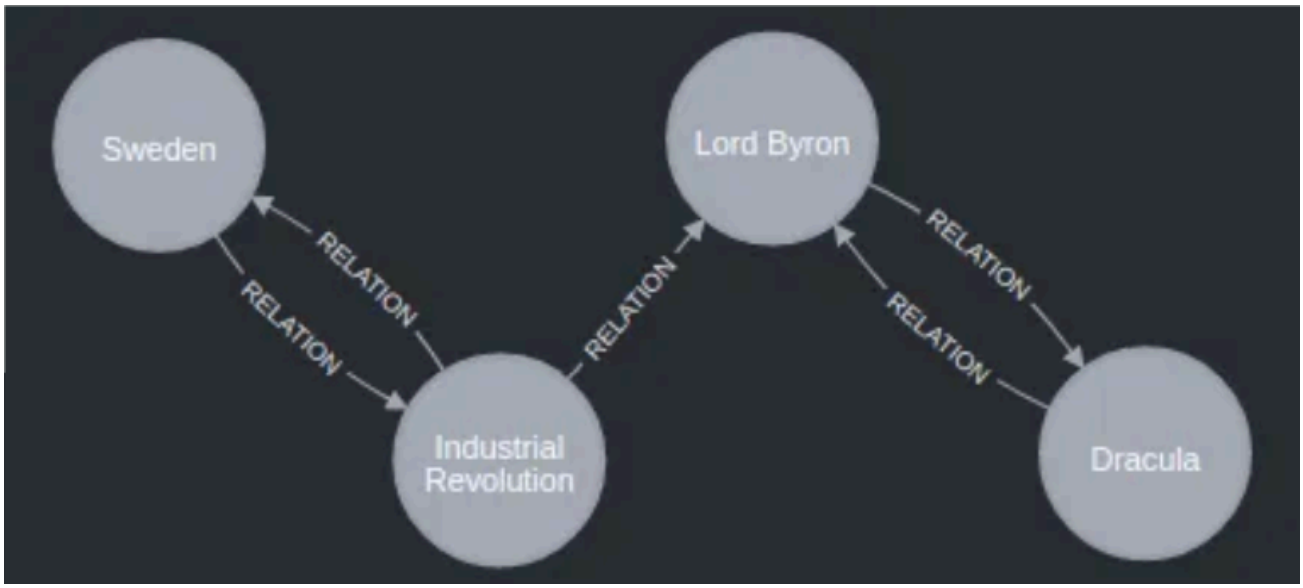
DAFTAR ISI

DAFTAR ISI 1
BAB I 2
DESKRIPSI TUGAS 2
1.1 Deskripsi Tugas 2
BAB II 3
2.1 Penjelajahan Graf 3
2.2 Algoritma BFS 3
2.3 Algoritma IDS & DLS 5
2.4 Website Jebir WikiRace 7
BAB III 8
3.1 Langkah-langkah Pemecahan Masalah 8
3.2 Proses Pemetaan Masalah 9
3.3 Tentang Website 11
3.4 Contoh Ilustrasi Kasus 11
3.4.1 Algoritma BFS 11
3.4.2 Algoritma IDS 13
BAB IV 16
4.1 Struktur Program 16
4.2 Backend 17
4.2.1 Struktur Data 17
4.2.2 Fungsi 17
4.2.4 Implementasi Algoritma 19
4.3 Frontend 30
4.4 Tata Cara Penggunaan Program 31
4.5 Hasil Pengujian 32
4.5.1 Antarmuka Website 32
4.5.2 Field Input 35
4.5.3 Uji Coba Kasus 37
4.6 Analisis Hasil Pengujian 46
BAB V 48
5.1 Kesimpulan 48
5.2 Saran 48
5.3 Refleksi 49
LAMPIRAN 50
DAFTAR PUSTAKA 51

BAB I
DESKRIPSI TUGAS

1.1 Deskripsi Tugas

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



(Sumber: https://miro.medium.com/v2/resize:fit:1400/1*jxmEbVn2FFWybZsIicJCWQ.png)

BAB II
LANDASAN TEORI

2.1 Penjelajahan Graf

Penjelajahan graf merupakan proses pengunjungan simpul demi simpul pada suatu graf untuk tujuan tertentu dengan cara yang sistematik. Graf sendiri adalah kumpulan simpul yang dihubungkan oleh sisi. Penjelajahan graf sering digunakan dalam pemodelan dan analisis sistem yang kompleks, seperti jaringan komputer, basis data, dan algoritma pencarian. Urutan penjelajahan graf menentukan tipe atau jenis penjelajahan graf. Metode yang dapat digunakan untuk melakukan penjelajahan graf contohnya adalah dengan menggunakan Breadth-First Search dan Iterative Deepening Search.

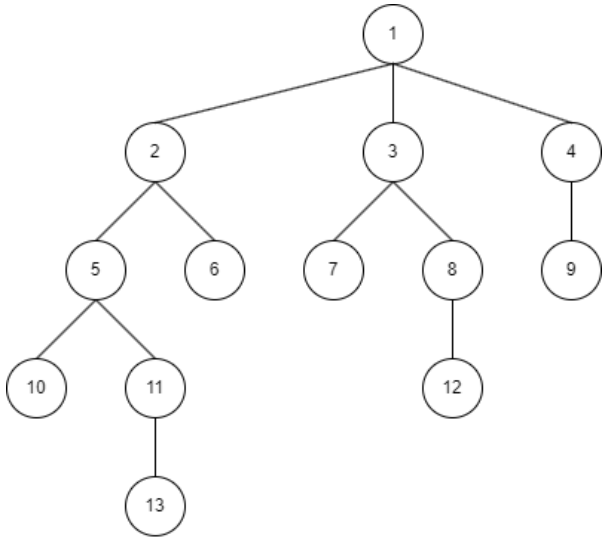
2.2 Algoritma BFS

Algoritma Breadth-First Search (BFS) merupakan metode penjelajahan graf dengan memulai penjelajahan dari simpul awal, dan berlanjut ke tetangga-tetangga dari simpul tersebut, sebelum mengunjungi simpul-simpul yang lebih jauh. Pada algoritma ini, umumnya dimanfaatkan struktur data *queue* untuk menyimpan simpul-simpul yang akan dikunjungi. Algoritma BFS menjamin bahwa simpul-simpul dikunjungi dalam urutan lebar, artinya simpul-simpul yang lebih dekat dengan simpul awal akan dikunjungi terlebih dahulu sehingga memungkinkan untuk menemukan jalur terpendek antara dua simpul pada graf tidak berarah dan tanpa bobot.

Misalkan, terdapat simpul V dari graf G. Penjelajahan graf dengan algoritma BFS memiliki skema umum penelusuran sebagai berikut.

1. Kunjungi simpul v.
2. Kunjungi seluruh simpul yang merupakan tetangga dari simpul v terlebih dahulu.
3. Kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang sebelumnya telah dikunjungi.
4. Ulangi langkah-langkah yang sama hingga simpul pada graf G telah dikunjungi seluruhnya.

Sebagai contoh, pada graf yang ditunjukkan pada Gambar 2.2.1 berikut dengan menggunakan algoritma BFS urutan simpul penjelajahan secara berurutan mulai dari simpul 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13.



Gambar 2.2.1 Urutan Algoritma BFS

Secara umum, pseudocode untuk algoritma BFS dalam penjelajahan graf adalah sebagai berikut.

```
procedure BFS(input v: simpul)
{ Melakukan traversal graf dengan algoritma pencarian BFS. Masukan: v adalah
simpul awal traversal Luaran: Seluruh simpul graf dijelajahi dengan algoritma
pencarian BFS }

Deklarasi
w : simpul
q : antrian
procedure Kunjungi(input/output w : simpul)
{ Mengunjungi simpul w }
function SudahDikunjungi(input w : w) → boolean
{ Mengecek apakah simpul w sudah dikunjungi }
procedure BuatAntrian(input/output q : antrian)
{ Membuat antrian kosong }
procedure MasukAntrian(input/output q : antrian, input v : simpul)
{ Memasukkan simpul v ke antrian q pada posisi belakang }
procedure HapusAntrian(input/output q : antrian, output v : simpul)
{ Menghapus v dari kepala antrian q }
function AntrianKosong(input q : antrian) → boolean
{ Mengecek apakah antrian q kosong }
```

```
Algoritma
BuatAntrian(q)
Kunjungi(v)
MasukAntrian(q,v)
{ Kunjungi semua simpul graf selama antrian belum kosong }
while not AntrianKosong(q) do
  HapusAntrian(q,v)
    for (tiap simpul w yang bertetangga dengan simpul v) do
      if not SudahDikunjungi(w) then
        Kunjungi(w)
        MasukAntrian(q,w)
      endif
    endfor
endwhile
{ AntrianKosong(q) }
```

2.3 Algoritma IDS & DLS

Algoritma Iterative Deepening Search (IDS) merupakan gabungan antara dua pendekatan yakni Depth-First Search (DFS) dan Depth-Limited Search (DLS) yang dilakukan secara iteratif sehingga algoritma ini memperoleh keuntungan dari DFS yang efisien dalam penggunaan memori dengan keuntungan dari DLS yang memastikan solusi ditemukan dalam kedalaman tertentu. Namun, IDS tidak selalu efisien dalam situasi di mana graf memiliki cabang yang sangat panjang atau ketika simpul pada level-level atas dibangkitkan berulang kali.

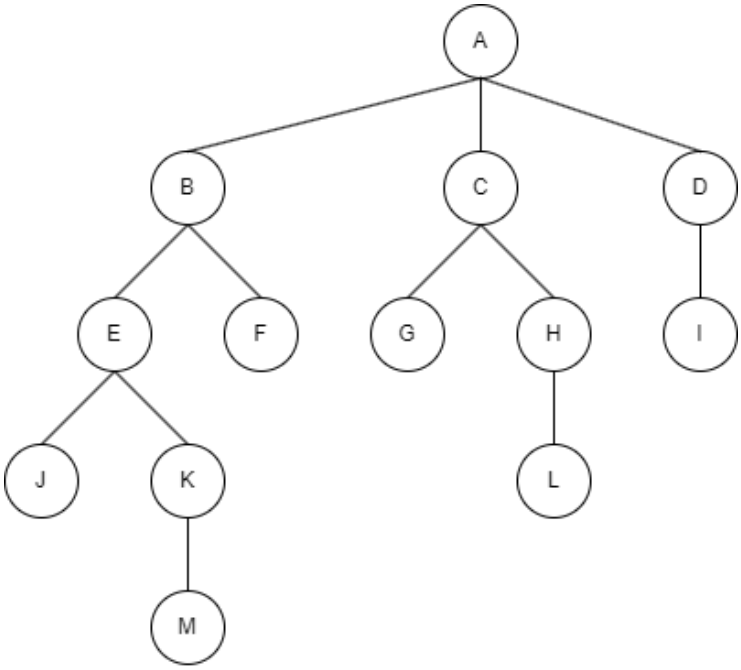
Misalkan, terdapat simpul *V* dari graf *G*. Penjelajahan graf dengan algoritma DLS serupa dengan penjelajahan graf dengan algoritma DFS namun dengan pembatasan kedalaman, misalnya l, sehingga simpul pada level l dianggap tidak memiliki successor. Skema umum penelusuran algoritma DLS adalah sebagai berikut.

1. Kunjungi simpul *v*
2. Kunjungi simpul *w* yang bertetangga dengan simpul *v*.
3. Ulangi DFS mulai dari simpul *w*.
4. Ketika mencapai simpul *u* sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi atau kedalaman telah mencapai kedalaman maksimal, pencarian

dirunut-balik (*backtrack*) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul w yang belum dikunjungi.

5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.

Pada algoritma IDS, penjelajahan dilakukan dengan melakukan serangkaian DFS seperti pada skema di atas, dengan peningkatan nilai kedalaman *cut-off*, hingga solusi ditemukan. Pada setiap iterasi diasumsikan bahwa simpul sebagian besar berada di level bawah sehingga tidak menjadi persoalan ketika simpul pada level-level atas dibangkitkan berulang kali.



Gambar 2.3.1 Graf G

Sebagai contoh, pada graf yang ditunjukkan pada Gambar 2.3.1 diatas dengan menggunakan algoritma IDS urutan simpul penjelajahan memiliki urutan setiap iterasinya sebagai berikut.

Tabel 2.3.1 Simpul Penjelajahan Algoritma IDS

Kedalaman	Urutan simpul
0	A
1	A, B, C, D
2	A, B, E, F, C, G, H, D, I

3	A, B, E, J, K, F, C, G, H, L, D, I
4	A, B, E, J, K, M, F, C, G, H, L, D, I

Secara umum, pseudocode untuk algoritma IDS dalam penjelajahan graf adalah sebagai berikut.

```
Function IDS (problem)
Depth ← 0
Iterate
  Result ← DLS(problem,depth)
stop: result ≠ cutoff
  depth depth + 1
→ result

Function DLS (problem, limit)
  → rec_DLS(make_node(init_state),problem,limit)

Function Rec_DLS (node,problem, limit)
  if isGoal(node) then → solution(node)
  else if depth(node)=limit then → cutoff
  else
    for each successor in Expand(node,problem) do
      result ß rec_DLS(successor,problem,limit)
      if result=cutoff then cutoff_occured ß true
      else if result≠failure then → result
  if cutoff_occured then → cutoff
  else → failure
```

2.4 Website Jebir WikiRace

Jebir WikiRace adalah sebuah website yang memungkinkan pengguna untuk melakukan pencarian artikel Wikipedia dari artikel awal ke artikel target menggunakan dua opsi pendekatan algoritma: Breadth-First Search (BFS) atau Iterative Deepening Search (IDS). Pendekatan algoritma ini diterapkan sebagai metode untuk melakukan penjelajahan graf di mana hipertaut artikel Wikipedia sendiri dapat direpresentasikan sebagai persoalan graf. Website ini dibangun dengan menggunakan bahasa pemrograman Go untuk logika pemrosesan data dan sebagai peran *backend*, serta ReactJS untuk antarmuka pengguna (*frontend*).

BAB III
ANALISIS PEMECAHAN MASALAH

3.1 Langkah-langkah Pemecahan Masalah

Persoalan utama dalam permainan WikiRace adalah untuk mendapatkan jalur terpendek dari artikel awal menuju ke artikel tujuan yang diberikan. Permasalahan ini dapat digambarkan sebagai sebuah graf dinamis atau graf yang terbentuk saat proses pencarian dilakukan. Pada graf dinamis ini, pembangkitan status barunya dilakukan dengan melakukan scraping link yang terdapat pada sebuah artikel Wikipedia. Hasil dari scraping akan menghasilkan sekumpulan link yang akan menjadi simpul tetangga baru. Link yang akan menjadi tetangga adalah link yang hanya mengarah menuju laman Wikipedia sehingga scraping link hanya akan diambil pada sebuah href yang mengandung string "/wiki". Selain itu, dilakukan juga filtering dalam scraping yaitu dengan tidak mengikutsertakan beberapa keyword di antaranya adalah "Main_Page", "Special:", "Help:", "Template:", "Category:", "Portal:", "Wikipedia:", "Talk:", dan "File:".

Dari pembangkitan graf dinamis tersebut, untuk melakukan pencarian dari artikel awal menuju artikel tujuan yang diberikan dapat dilakukan penjelajahan graf dengan menggunakan algoritma BFS dan IDS.

3.1.1 Algoritma BFS

Pemecahan masalah WikiRace menggunakan algoritma BFS dilakukan dengan langkah-langkah sebagai berikut.

1. Pada kasus di mana artikel awal yang dimasukkan adalah sama dengan artikel tujuan, maka jalur dari artikel awal menuju artikel tujuan langsung ditemukan, jumlah artikel yang diperiksa berjumlah 1, dan jumlah artikel yang diklik adalah 0.
2. Jika tidak, maka akan dilakukan proses BFS. Masukkan artikel awal sebagai node dengan kedalaman 0 dan tanpa parent ke dalam antrian (*queue*) untuk dilakukan penelusuran.
3. Ambil dan keluarkan elemen paling depan dari *queue*, dalam hal ini tidak perlu dilakukan pengecekan kesamaan artikel kembali karena artikel yang masuk ke dalam *queue* bukan merupakan artikel tujuan. Jika artikel sudah pernah dikunjungi maka ulangi lagi langkah ketiga ini dari awal. Jika belum, berikan markah bahwa artikel saat ini sudah pernah dikunjungi, lalu lakukan scraping link yang masih mengarah ke laman wikipedia.
4. Pada setiap link hasil scraping akan dilakukan pengecekan. Jika link belum pernah dikunjungi, tambahkan jumlah artikel yang diperiksa. Jika link belum pernah dikunjungi dan link tersebut merupakan artikel tujuan, maka solusi ditemukan dengan jumlah artikel yang diklik merupakan kedalaman node saat ini ditambah 1. Namun, Jika link belum pernah dikunjungi dan bukan merupakan artikel tujuan, maka

tambahkan link tersebut sebagai node dengan kedalamannya adalah node saat ini ditambah 1 dan node parentnya adalah node saat ini.

5. Ulangi kembali langkah ketiga hingga solusi ditemukan. Jika ingin mencari keseluruhan jalur yang mungkin, meskipun jalur telah ditemukan, tetap lakukan pencarian dengan melakukan penjelajahan seperti pada langkah-langkah tersebut dengan memberikan batasan ekspansi nodenya tidak melebihi kedalaman dari saat pertama kali solusi ditemukan.

3.1.1 Algoritma IDS

Pemecahan masalah WikiRace menggunakan algoritma BFS dilakukan dengan langkah-langkah sebagai berikut.

1. Inisialisasi iterasi kedalaman pencarian (depth) dengan 0.
2. Pada setiap iterasi, implementasikan DLS untuk mencari jalur dengan batasan kedalaman tertentu.
    a. DLS melakukan pencarian dari simpul saat ini ke tetangga-tetangganya dengan membatasi kedalaman pencarian secara rekursif.
    b. Saat mencapai simpul daun, akan diperiksa apakah URL saat ini sama dengan target URL. Jika ya, maka kembalikan jalur yang ditemukan. Jika tidak, maka pencarian berhenti dengan tidak ditemukan jalur menuju artikel tujuan. Catat artikel yang diperiksa menjadi bertambah 1 jumlahnya.
    c. Jika belum mencapai simpul daun, maka akan dilakukan scraping link untuk melakukan ekspansi simpul-simpul. Tandai artikel saat ini menjadi telah dikunjungi untuk menghindari duplikasi pengunjungan.
    d. Setiap tetangga yang belum dikunjungi akan dieksplorasi dengan memanggil rekursif fungsi DLS dengan jarak menuju simpul daun yang berkurang.
    e. Jika jalur menuju target ditemukan, jalur tersebut akan dikembalikan bersama dengan jumlah artikel yang diperiksa.
3. Jika pada akhir DLS tidak ditemukan jalur, lakukan kembali DLS dengan meningkatkan kedalamannya (increment 1). Jika setelah 5 menit dan DLS selesai namun belum ditemukan, maka pencarian akan berhenti dilakukan.
4. Jika ingin mencari keseluruhan jalur yang mungkin, meskipun jalur telah ditemukan, tetap lakukan pencarian dengan melakukan penjelajahan seperti pada langkah-langkah tersebut dengan memberikan batasan ekspansi nodenya tidak melebihi kedalaman dari saat pertama kali solusi ditemukan.

3.2 Proses Pemetaan Masalah

Persoalan WikiRace dapat diselesaikan menggunakan algoritma BFS dan IDS. Pemetaan dilakukan dengan memandang persoalan ini sebagai sebuah pohon yang akan dilakukan penjelajahan dari *initial state* tertentu menuju ke *goal state* yang ditentukan. Pohon pada persoalan ini merupakan pohon dinamis yang pembangkitan statusnya dilakukan dengan mengambil setiap tautan ke artikel

lain yang masih menuju ke laman Wikipedia. Proses pemetaan persoalan WikiRace menjadi elemen-elemen pohon yang dinamis adalah sebagai berikut.

1. Ruang status (*state space*): himpunan semua artikel Wikipedia yang dapat dijangkau dari artikel awal. Setiap simpul dalam pohon ruang status mewakili satu artikel Wikipedia.
2. Akar (*initial State):* simpul awal dari pencarian, yang merupakan artikel awal yang diberikan.
3. Daun (*goal state*): simpul yang mencapai artikel tujuan atau simpul yang tidak memiliki simpul ekspansi lagi untuk dieksplorasi.
4. Operator: penambahan simpul dilakukan dengan mengeksplorasi simpul-simpul yang terhubung langsung dari simpul saat ini. Tautan-tautan yang ada di dalam artikel Wikipedia spesifik akan menjadi daftar simpul hasil ekspansi yang akan dieksplorasi pada langkah berikutnya sesuai algoritma yang digunakan.
5. Status persoalan (*problem state*): posisi saat ini dalam permainan, termasuk artikel Wikipedia yang sedang diperiksa dan semua artikel yang telah diperiksa sebelumnya.
6. Status solusi (*solution state*): satu atau lebih status yang menyatakan solusi persoalan berupa artikel Wikipedia yang telah dituju. Pencarian berakhir ketika simpul solusi ditemukan.
7. Ruang solusi (*solution space*): himpunan semua status solusi berupa himpunan semua artikel Wikipedia yang dapat dijangkau dari artikel awal dan menjadi artikel tujuan.

3.2.1 Pemetaan ke Algoritma BFS

BFS adalah algoritma pencarian yang mengeksplorasi semua simpul pada level yang sama sebelum melanjutkan ke level berikutnya. Pemetaan persoalan WikiRace ke BFS adalah sebagai berikut:

1. *Queue*: BFS menggunakan struktur data antrian (*queue*) untuk menyimpan simpul yang akan dieksplorasi selanjutnya. Dalam konteks WikiRace, akan disimpan tautan-tautan dari artikel yang akan diperiksa selanjutnya dalam antrian.

3.2.1 Pemetaan ke Algoritma IDS

IDS merupakan variasi dari Depth-First Search (DFS) yang mencari solusi secara bertahap dengan meningkatkan kedalaman pencarian pada setiap iterasi. Pemetaan persoalan WikiRace ke IDS adalah sebagai berikut:

1. Depth Limit: depth limit membatasi kedalaman pencarian pada setiap iterasi. Dalam konteks WikiRace, kedalaman ini bisa dianggap sebagai jumlah langkah maksimum yang diambil untuk mencapai artikel tujuan.
2. Iterasi: IDS melakukan pencarian secara bertahap dengan meningkatkan kedalaman pencarian pada setiap iterasi. Pada setiap iterasi, dilakukan pencarian dengan kedalaman tertentu.
3. *Backtracking*: IDS memanfaatkan *backtracking* untuk kembali ke node sebelumnya jika pencarian tidak menghasilkan solusi. Dalam WikiRace, *backtracking* digunakan untuk mencoba jalur alternatif jika jalur saat ini tidak berhasil mencapai artikel tujuan.

3.3 Tentang Website

Fitur fungsional pada aplikasi web yang dibangun utamanya adalah melakukan pencarian rute terpendek dari satu artikel ke artikel lainnya pada hipertaut Wikipedia. Hal ini dilakukan dengan memanfaatkan algoritma BFS (Breadth First Search) dan DFS (Depth First Search) yang telah diimplementasikan pada backend aplikasi. Pengguna dapat memasukkan artikel awal dan artikel tujuan yang ingin dicari rute terpendeknya, serta memilih algoritma yang ingin digunakan. Setelah melakukan pencarian, pengguna akan mendapatkan informasi berupa rute terpendek yang ditemukan, jumlah artikel yang telah dicek, jumlah artikel yang diklik, dan waktu eksekusi yang dibutuhkan untuk menemukan rute tersebut.

Untuk menunjang fungsionalitas utama tersebut, pada bagian input dibuat fitur autocomplete pada input artikel yang memungkinkan pengguna untuk memilih artikel dari sugesti yang muncul saat pengguna mengetikkan judul artikel. Terdapat pula fitur untuk menukar judul artikel awal dan tujuan dengan satu klik. Selain itu, pengguna dapat memilih algoritma yang ingin digunakan dengan mudah melalui dropdown yang disediakan. Pada bagian output hasil, waktu proses pencarian mungkin tidak instan sehingga pengguna perlu menunggu. Dibuat fitur *fun fact* yang muncul saat proses pencarian sedang berlangsung. Selain itu, pada bagian output juga dibuat fitur untuk menampilkan artikel Wikipedia beserta artikel-artikel yang terhubung dengan artikel tersebut. Pengguna dapat mengunjungi artikel tersebut dengan menekan klik untuk membaca lebih lanjut pada tab baru.

Arsitektur web yang dibangun terdiri atas dua bagian utama, yaitu backend dan frontend. Backend dibangun menggunakan bahasa pemrograman Go dan frontend menggunakan bahasa pemrograman JavaScript dengan framework React. Backend bertugas untuk melakukan pencarian rute terpendek dari satu artikel ke artikel lainnya pada hipertaut Wikipedia. Backend menerima data input berupa artikel awal, artikel tujuan, dan algoritma yang ingin digunakan dari frontend, kemudian melakukan pencarian rute terpendek dan mengirimkan data output ke frontend. Frontend bertugas untuk menampilkan antarmuka aplikasi web yang memungkinkan pengguna untuk memasukkan data input dan menampilkan data output yang diterima dari backend. Frontend juga bertugas untuk menampilkan artikel Wikipedia yang telah dicari rute terpendeknya beserta artikel-artikel yang terhubung dengan artikel tersebut.

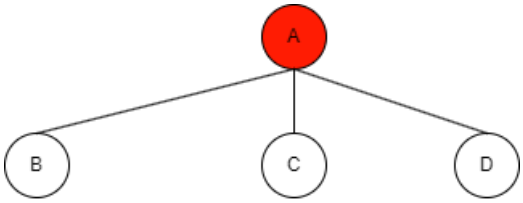
3.4 Contoh Ilustrasi Kasus

Misalkan akan dicari jalur dari sebuah artikel awal A menuju ke artikel tujuan K, maka pembangkitan simpul-simpul ekspansi pada masing-masing algoritma BFS dan IDS akan berbeda.

3.4.1 Algoritma BFS

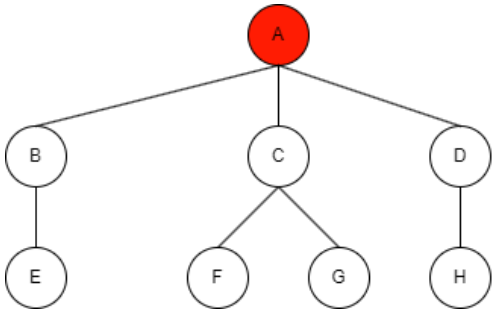
Pada algoritma BFS, prosesnya yaitu menggunakan queue untuk menyimpan simpul hidup. Pencarian dilakukan berdasarkan lebarnya terlebih dahulu sebelum berganti kedalaman. Jika akan dicari jalur dari link A ke link K:

Pertama cari simpul tetangga dari link A dan masukan ke dalam queue. Simpul simpul tetangga ini nantinya akan dicari tetangganya juga.



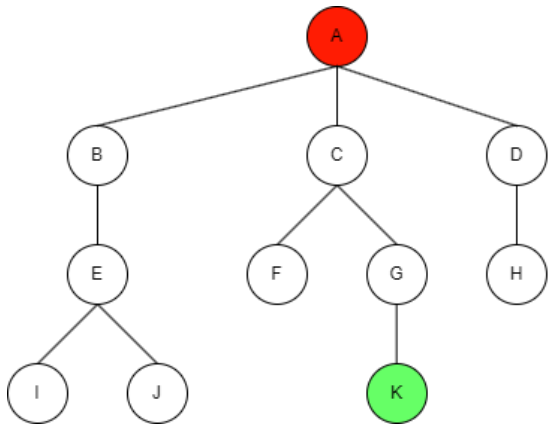
Simpul-E	Simpul hidup
A	B_A, C_A, D_A

Setelah itu ambil elemen paling depan dari simpul hidup dan cari link tetangga dari link yang diambil itu sampai mendapatkan link tujuan



Simpul-E	Simpul hidup
A	B_A, C_A, D_A
B_A	C_A, D_A, E_{AB}
C_A	D_A, E_{AB}, F_{AC}, G_{AC}
D_A	E_{AB}, F_{AC}, G_{AC}, H_{AD},

Sampai sejauh ini belum ditemukan link tujuan, maka akan dicari lagi sampai dapat.



Simpul-E	Simpul hidup
A	B_A, C_A, D_A
B_A	C_A, D_A, E_{AB}
C_A	D_A, E_{AB}, F_{AC}, G_{AC}
D_A	E_{AB}, F_{AC}, G_{AC}, H_{AD}
E_{AB}	F_{AC}, G_{AC}, H_{AD}, I_{ABE}, J_{ABE}
F_{AC}	G_{AC}, H_{AD}, I_{ABE}, J_{ABE},
G_{AC}	H_{AD}, I_{ABE}, J_{ABE}, K_{ACG}
H_{AD}	I_{ABE}, J_{ABE}, K_{ACG}
I_{ABE}	J_{ABE}, K_{ACG}
J_{ABE}	K_{ACG}
K_{ACG}	Solusi Ketemu

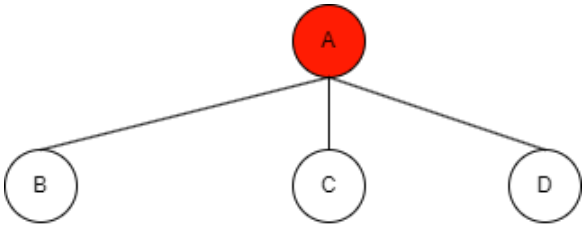
Pada link F,I,J,H sudah tidak memiliki link tetangga. Sehingga link tersebut dihapus dari queue namun tidak menambahkan link apapun ke dalam queue.
Solusi juga ditemukan dengan path A,C,G,K

3.4.2 Algoritma IDS

Pada algoritma IDS, prosesnya yaitu melakukan pencarian secara iterative yang meningkat berdasarkan kedalaman dengan memanggil DLS (Depth-Limited Search) pada tiap kedalaman. DLS merupakan modifikasi atau perpanjangan dari DFS (Depth First Search) yang mengimplementasikan batas untuk kedalamannya. Pada program yang dibuat, DLS

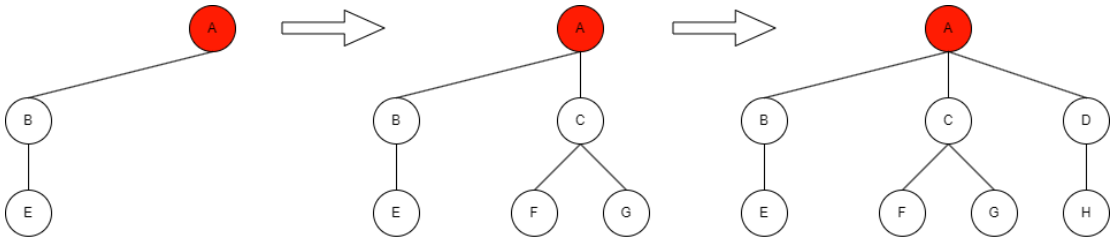
diimplementasi secara rekursif dengan memanggil lagi fungsi DLS itu sendiri. Jika akan dicari jalur dari link A ke simpul link K:

Pertama, program akan cek pada kedalaman 0 dengan memanggil DLS dengan maksimal depth 0. Pada kedalam ini hanya dihasilkan 1 link yaitu link start itu sendiri (link A). Setelah itu, program akan melanjutkan ke kedalaman berikutnya yaitu kedalaman 1. Pada kedalaman ini, dipanggil lagi DLS dengan kedalaman 1 yaitu link start dan simpul-simpul tetangganya. Tetangga tetangga ini didapatkan dari hasil scraping link. Didapatkan graf sebagai berikut:



Dengan urutan pencariannya: A, B , C, D.

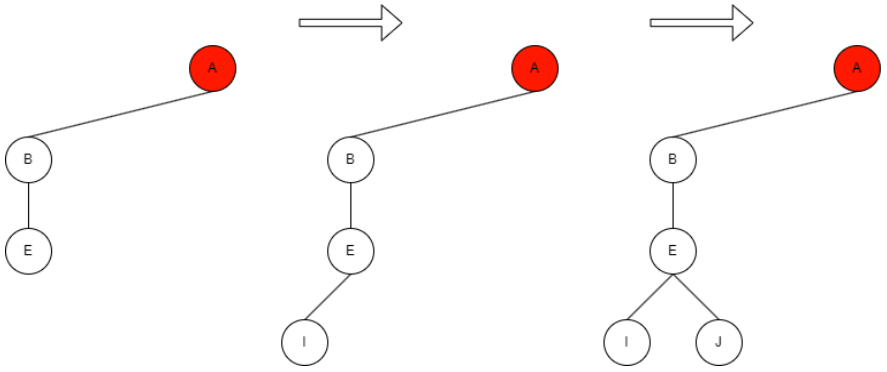
Karena link K belum ditemukan, maka proses iterasi berlanjut ke kedalaman 2. Proses diulangi dari awal di kedalaman 0 hingga ke kedalam 2. Pada kedalam 1 program akan mendapatkan tetangga yaitu simpul B, C, dan D. Program akan melanjutkan proses DLS terhadap simpul link B terlebih dahulu lalu ke link C lalu yang terakhir ke link D. Didapatkan graf sebagai berikut:



Dengan urutan pencariannya: A, B, E, C, F, G, D, dan H

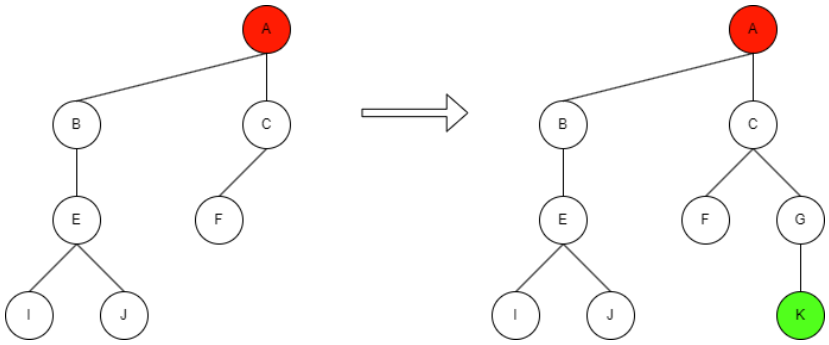
Pada iterasi di kedalaman 2, belum juga ditemukan link K sebagai link tujuannya. Maka, program akan menjalankan lagi DLS sampai ke kedalam ke 3. Prosesnya program akan melakukan rekursif pemanggilan DLS. Pertama program melakukan DLS link A, lalu mendapatkan link tetangga B, C, dan D. Lakukan DLS pada link B sampai selesai yaitu sebagai berikut:

- Didapat tetangga E. Lalu lakukan DLS pada link E dan didapat link tetangga yaitu I, J
- Karena pada link I dan J sudah ada pada kedalaman 3, maka proses DLS pada link B selesai



Setelah itu, lakukan DLS pada link C sampai selesai yaitu sebagai berikut:

- Didapat tetangga F dan G
- Lalu lakukan DLS pada link F, namun karena link F tidakmemiliki tetangga, DLS dilanjutkan ke link G
- Pada DLS link G didapatkan tetangga K. Karena K merupakan link tujuan, maka proses ini akan dihentikan



Pada iterasi ketiga ini, program check artikel dengan urutan berikut A, B, E, I, J, C, F, G, K. Namun, untuk output path dari link awal ke link tujuan yaitu A, C, G, K.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1 Struktur Program

```
Tubes2_Jebir/
├── doc/
├── test/
├── src/
│   ├── backend/
│   │   ├── lib/
│   │   │   ├── back.go
│   │   │   ├── BFS.go
│   │   │   ├── BFSAllPath.go
│   │   │   ├── BFSgorut.go
│   │   │   ├── IDS.go
│   │   │   ├── IDSAllPath.go
│   │   │   └── util.go
│   │   ├── main.go
│   │   ├── go.mod
│   │   ├── go.sum
│   │   └── Dockerfile
│   │
│   ├── frontend/
│   │   ├── node_modules
│   │   ├── src/
│   │   │   ├── components/
│   │   │   │   ├── assets/
│   │   │   │   ├── About.js
│   │   │   │   ├── FormInput.js
│   │   │   │   ├── GraphVisualization.js
│   │   │   │   ├── Header.js
│   │   │   │   ├── HowToUse.js
│   │   │   │   └── Result.js
│   │   │   ├── App.js
│   │   │   ├── Home.js
│   │   │   └── index.js
│   │   ├── Dockerfile
│   │   ├── package-lock.json
│   │   └── package.json
│   └── Docker-compose.yml
└── README.md
```

Program ini dibagi menjadi dua bagian utama yaitu frontend dan backend. Backend program ini berfungsi untuk menerima data dari frontend, melakukan pemrosesan data, dan mengirimkan hasil pemrosesan data kembali ke frontend. Pada folder lib di backend ini berisi logika utama untuk algoritma BFS, IDS, utilitas untuk scraping dan membentuk sebuah path. Sementara frontend program ini berfungsi untuk mengirimkan data ke backend, menerima hasil pemrosesan data dari backend, dan menampilkan hasil pemrosesan data kepada pengguna. Untuk memudahkan, digunakan Docker yang dapat langsung mengintegrasikan keduanya.

4.2 Backend

4.2.1 Struktur Data

Dalam program ini terutama pada bagian backend, digunakan struktur data berupa FormData dan Node.

1. **FormData**
   FormData adalah struktur data yang digunakan untuk menyimpan data yang diterima dari request HTTP. Struktur data ini memiliki beberapa field, yaitu StartArticle, StartUrl, EndArticle, EndUrl, dan Algoritma. Field-field ini merepresentasikan data yang diterima dari request HTTP, yaitu nama artikel awal, URL artikel awal, nama artikel tujuan, URL artikel tujuan, dan algoritma pencarian yang digunakan.
2. **Node**
   Node adalah struktur data yang digunakan untuk merepresentasikan simpul dalam algoritma BFS dan IDS. Struktur data ini memiliki tiga field, yaitu Depth, Url, dan Prev. Field Depth merepresentasikan kedalaman node, field Url merepresentasikan URL node, dan field Prev merepresentasikan node sebelumnya.

4.2.2 Fungsi

Sementara untuk fungsi yang dibangun guna menunjang keberjalanan algoritma dan program di antaranya adalah:

1. **MainBackend**
   MainBackend adalah fungsi utama yang digunakan untuk menjalankan algoritma BFS atau IDS berdasarkan data yang diterima dari request HTTP. Fungsi ini menerima data FormData dan mengembalikan path, jumlah artikel yang diperiksa, jumlah artikel yang diklik, waktu eksekusi, dan error. Fungsi ini akan menjalankan algoritma BFS jika algoritma yang digunakan adalah BFS, dan menjalankan algoritma IDS jika algoritma yang digunakan adalah IDS.
2. **BFS**
   BFS adalah fungsi yang digunakan untuk menjalankan algoritma Breadth-First Search (BFS) untuk mencari rute dari artikel awal ke artikel tujuan. Fungsi ini

menerima URL artikel awal dan URL artikel tujuan, dan mengembalikan path, jumlah artikel yang diperiksa, jumlah artikel yang diklik, dan error. Sesuai namanya, fungsi ini akan melakukan pencarian rute dari artikel awal ke artikel tujuan menggunakan implementasi algoritma BFS.

3. **BFSAllPath**
   Secara umum, BFSAllPath sama seperti algoritma BFS. Yang membedakan yaitu jika BFS hanya untuk mencari 1 path terpendek saja, BFSAllPath digunakan untuk mencari semua path dengan panjang paling pendek.
4. **IDS**
   IDS adalah fungsi yang digunakan untuk menjalankan algoritma Iterative Deepening Search (IDS) untuk mencari rute dari artikel awal ke artikel tujuan. Fungsi ini menerima URL artikel awal dan URL artikel tujuan, dan mengembalikan path, jumlah artikel yang diperiksa, jumlah artikel yang diklik, dan error. Fungsi ini juga akan melakukan pencarian rute dari artikel awal ke artikel tujuan namun dengan menggunakan algoritma IDS.
5. **IDSAllPath**
   Secara umum, IDSAllPathsama seperti algoritma IDS. Yang membedakan yaitu jika IDS hanya untuk mencari 1 path terpendek saja, IDSAllPath digunakan untuk mencari semua path dengan panjang paling pendek.
6. **scrapeLinks**
   scrapeLinks adalah fungsi yang digunakan untuk melakukan scraping link dari URL yang diberikan. Fungsi ini menerima URL dan mengembalikan array of string yang berisi link yang ditemukan. Pada fungsi ini, dimanfaatkan library goquery untuk melakukan scraping link dari URL wikipedia yang diberikan dengan cara membuat request HTTP ke URL tersebut dan melakukan parsing HTML menggunakan goquery.
7. **buildPathToTarget**
   buildPathToTarget adalah fungsi yang digunakan untuk membangun path dari node target ke node awal. Fungsi ini menerima node target dan mengembalikan path dari node target ke node awal.

4.2.3 File pada Backend

a. **main.go**
   main.go, berfungsi sebagai file utama yang akan dijalankan ketika program backend dijalankan. File ini berisi fungsi main yang akan menjalankan server backend dan menangani request yang diterima dari frontend. File ini juga berisi fungsi handleInsert yang akan menangani request POST yang diterima dari frontend dan memanggil fungsi MainBackend dari package backend untuk melakukan pemrosesan data. Setelah pemrosesan data selesai, fungsi handleInsert akan mengirimkan hasil pemrosesan data kembali ke frontend.

b. **back.go**
back.go, berisi definisi struct FormData yang digunakan untuk menyimpan data yang diterima dari frontend dan struct Node yang digunakan. File ini juga berisi fungsi MainBackend yang mengarahkan pemrosesan data ke algoritma yang sesuai berdasarkan data yang diterima dari frontend. Fungsi ini akan mengembalikan hasil pemrosesan data ke fungsi handleInsert.

c. **BFS.go**
BFS.go, berisi fungsi BFS yang digunakan untuk melakukan pencarian rute terpendek dari artikel awal ke artikel tujuan menggunakan algoritma BFS. Pada bagian ini sudah diterapkan goroutine untuk membantu mempercepat proses pencarian rute.

d. **BFSAllPath.go**
BFSAllPath.go, berisi fungsi BFSAllPath yang digunakan untuk melakukan pencarian semua rute yang mungkin dari artikel awal ke artikel tujuan menggunakan algoritma BFS.

e. **IDS.go**
IDS.go, berisi fungsi IDS yang digunakan untuk memproses masukan URL artikel awal dan artikel tujuan, kemudian melakukan pencarian rute terpendek dari artikel awal ke artikel tujuan menggunakan algoritma IDS.

f. **IDSAllPath.go**
IDSAllPath.go, berisi fungsi IDSAllPath yang digunakan untuk melakukan pencarian semua rute yang mungkin dari artikel awal ke artikel tujuan menggunakan algoritma IDS.

g. **util.go**
util.go, berisi fungsi-fungsi utilitas yang digunakan oleh algoritma BFS dan IDS yakni fungsi buildPathToTarget, fungsi scrapeLinks, dan fungsi filter yang dipergunakan oleh scraper untuk memfilter link yang diinginkan.

4.2.4 Implementasi Algoritma

BFS

```go
func BFS(start, end string) ([]string, int, int, error) {
    //menangani kasus nil
    if len(start) == 0 || len(end) == 0 {
        return nil, 0, 0, nil
    }

    checkedArticle := 1
    if start == end { //jika start == end langsung return
        return []string{start}, checkedArticle, 0, nil
    }
```

```go
    visited := make(map[string]bool)
    queue := []*Node{{Url: start, Depth: 0}}

    for len(queue) > 0 { //jika queue belum 0 maka loop terus
        currentNode := queue[0]
        queue = queue[1:]

        if visited[currentNode.Url] {
            continue
        }

        visited[currentNode.Url] = true

        links, err := scrapeLinks(currentNode.Url) //mencari link tetangga
        if err != nil {
            return nil, checkedArticle, 0, err
        }

        clickArticle := currentNode.Depth + 1
        for _, link := range links {
            if !visited[link] {
                checkedArticle++
                if link == end {
                    return append(buildPathToTarget(currentNode), link),
checkedArticle, clickArticle, nil
                }
                //menambahkan link tetangga di queue
                queue = append(queue, &Node{Url: link, Prev: currentNode,
Depth: currentNode.Depth + 1})
            }
        }
    }
    return nil, checkedArticle, 0, fmt.Errorf("not found")
}
```

IDS

```
func DLS(URL string, target string, depth int, visited map[string]bool)
([]string, int, error) {
    artikel_diperiksa := 0
    if depth == 0 && URL == target {
        artikel_diperiksa++
        return []string{URL}, artikel_diperiksa, nil
    }

    if depth == 0 {
        artikel_diperiksa++
    }

    if depth > 0 {
        visited[URL] = true
        neighbors, err := scrapeLinks(URL) //mencari link tetangga
        if err != nil {
            return nil, 0, err
        }
        for _, neighbor := range neighbors {
            if !visited[neighbor] {
                temp, count, err2 := DLS(neighbor, target, depth-1,
visited)
                if err2 != nil {
                    return nil, 0, err2
                }
                artikel_diperiksa += count
                if temp != nil {
                    result := append([]string{URL}, temp...)
                    return result, artikel_diperiksa, nil
                }
            }
        }
    }

    return nil, artikel_diperiksa, nil
}
```

```go
func IDS(startURL string, targetURL string) ([]string, int, int, error) {
    //menangani kasus nil
    if len(startURL) == 0 || len(targetURL) == 0 {
        return nil, 0, 0, nil
    }

    artikel_diperiksa := 0
    startTime := time.Now()
    //melakukan pencarian untuk tiap kedalaman
    for depth := 0; time.Since(startTime).Seconds() <= 300; depth++ {
        visited := make(map[string]bool)

        result, count, err := DLS(startURL, targetURL, depth, visited)
        artikel_diperiksa += count
        if err != nil {
            return nil, artikel_diperiksa, 0, err
        }
        if result != nil {
            return result, artikel_diperiksa, len(result) - 1, nil
        }
    }
    return nil, artikel_diperiksa, 0, fmt.Errorf("tidak ditemukan path")
}
```

BFSAllPath

```go
func BFSAllPath(start, end string) ([][]string, int, int, error) {
    //menangani kasus nil
    if len(start) == 0 || len(end) == 0 {
        return nil, 0, 0, nil
    }

    checkedArticle := 1
    clickedArticle := 1
    var paths [][]string
    if start == end { //jika start == end langsung return
```

```
        path := []string{start}
        return append(paths, path), checkedArticle, clickedArticle, nil
    }

    visited := make(map[string]bool)
    visitedPaths := make(map[string]bool)
    queue := []*Node{{Url: start, Depth: 0}}
    depth := 9999
    for len(queue) > 0 { //jika queue belum 0 maka loop terus
        currentNode := queue[0]
        queue = queue[1:]

        if !visited[currentNode.Url] {
            visited[currentNode.Url] = true
            if currentNode.Depth < depth {
                links, err := scrapeLinks(currentNode.Url)
                if err != nil {
                    return nil, checkedArticle, 0, err
                }

                for _, link := range links {
                    if !visited[link] {
                        checkedArticle++
                        if link == end {
                            newPath := append(buildPathToTarget
(currentNode), link)
                            pathString := strings.Join(newPath, "->")
                            if !visitedPaths[pathString] {
                                visitedPaths[pathString] = true
                                depth = currentNode.Depth + 1
                                paths = append(paths, newPath)
                            }
                        }
                        if currentNode.Depth+1 < depth {
                            //menambahkan link tetangga di queue
                            queue = append(queue, &Node{Url: link, Prev:
currentNode, Depth: currentNode.Depth + 1})
                        }
                    }
```

```
                }
            }
        }
    }
    if len(paths) == 0 {
        return nil, checkedArticle, 0, fmt.Errorf("not found")
    }
    return paths, checkedArticle, depth, nil
}
```

IDSAllPath

```
func DLSAllPath(URL string, target string, depth int, visited
map[string]bool, path []string, visitedPath map[string]bool, allPaths
*[][]string, artikelDiperiksa *int) error {
    if depth == 0  && URL == target{
        *artikelDiperiksa++
        newPath := append([]string(nil), path...)
        pathString := strings.Join(newPath, "->")
        if !visitedPath[pathString] {
            visitedPath[pathString] = true
            *allPaths = append(*allPaths, newPath)
            return nil
        }
    }

    if depth == 0 {
        *artikelDiperiksa++
    }

    if depth > 0 {
        visited[URL] = true
        defer delete(visited, URL)

        neighbors, err := scrapeLinks(URL) //mencari link tetangga
        if err != nil {
            return err
```

```
        }

        for _, neighbor := range neighbors {
            if !visited[neighbor] {
                newPath := append(path, neighbor)
                err := DLSAllPath(neighbor, target, depth-1, visited,
newPath, visitedPath, allPaths, artikelDiperiksa)
                if err != nil {
                    return err
                }
            }
        }
    }

    return nil
}

func IDSAllPath(startURL string, targetURL string) ([][]string, int, int,
error) {
    //menangani kasus nil
    if len(startURL) == 0 || len(targetURL) == 0 {
        return nil, 0, 0, nil
    }

    artikelDiperiksa := 0
    allPaths := [][]string{}
    startTime := time.Now()
    //melakukan pencarian untuk tiap kedalaman
    for depth := 0; time.Since(startTime).Seconds() <= 300; depth++ {
        //inisialisasi
        visited := make(map[string]bool)
        visitedPath := make(map[string]bool)
        path := []string{startURL}

        err := DLSAllPath(startURL, targetURL, depth, visited, path,
visitedPath, &allPaths, &artikelDiperiksa)
        if err != nil {
            return nil, artikelDiperiksa, depth, err
        }
```

```
        if len(allPaths) > 0 {
            return allPaths, artikelDiperiksa, depth, nil
        }
    }

    return nil, artikelDiperiksa, 0, fmt.Errorf("tidak ditemukan path")
}
```

BFSgorut

```
func BFSgorut(start, end string) ([]string, int, int, error) {
    if len(start) == 0 || len(end) == 0 {
        return nil, 0, 0, nil
    }

    checkedArticle := 1
    if start == end {
        return []string{start, end}, checkedArticle, 0, nil
    }
    visited := make(map[string]bool)
    queue := make(chan *Node, 40000000)

    var wg sync.WaitGroup
    var visitedMutex sync.Mutex

    found := false
    path := []string{}
    clickArticle := 0

    queue <- &Node{Url: start, Depth: 0}
    startNode := <-queue
    visited[startNode.Url] = true

    links, err := scrapeLinks(startNode.Url)
    if err != nil {
        return nil, checkedArticle, 0, err
    }
```

```
    clickArticle = startNode.Depth + 1
    for _, link := range links {
        if !visited[link] {
            checkedArticle++
            if link == end {
                path = append(buildPathToTarget(startNode), link)
                found = true
                break
            }
            queue <- &Node{Url: link, Prev: startNode, Depth:
startNode.Depth + 1}
        }
    }

    for len(queue) > 0 && !found {
        currentNode := <-queue

        time.Sleep(5 * time.Millisecond)

        wg.Add(1)

        go func(currentNode *Node) {
            defer wg.Done()
            visitedMutex.Lock()
            if visited[currentNode.Url] {
                visitedMutex.Unlock()
                return
            }
            visitedMutex.Unlock()

            visitedMutex.Lock()
            visited[currentNode.Url] = true
            visitedMutex.Unlock()

            links, err := scrapeLinks(currentNode.Url)
            if err != nil {
                path = nil
                return
```

```
            }

            clickArticle = currentNode.Depth + 1
            for _, link := range links {
                visitedMutex.Lock()
                if !visited[link] {
                    checkedArticle++
                    if link == end {
                        path = append(buildPathToTarget(currentNode), link)
                        found = true
                        return
                    }
                    queue <- &Node{Url: link, Prev: currentNode, Depth:
currentNode.Depth + 1}
                }
                visitedMutex.Unlock()
            }
        }(currentNode)
    }

    if found {
        return path, checkedArticle, clickArticle, nil
    }

    wg.Wait()
    close(queue)

    return nil, checkedArticle, 0, fmt.Errorf("not found")
}
```

scrapeLinks

```
func filter(str string, keywords []string) bool {
    for _, keyword := range keywords {
        if strings.Contains(str, keyword) {
            return true
        }
    }
```

```
    return false
}

func scrapeLinks(url string) ([]string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()

    doc, err := goquery.NewDocumentFromReader(resp.Body)
    if err != nil {
        return nil, err
    }

    var links []string
    filterWords := []string{"Main_Page", "Special:", "Help:", "Template:",
"Category:", "Portal:", "Wikipedia:", "Talk:", "File:"}
    doc.Find("a").Each(func(i int, s *goquery.Selection) {
        href, exists := s.Attr("href")
        parts := strings.Split(href, "#")
        result := parts[0]
        if exists && strings.HasPrefix(result, "/wiki/") && !filter(result,
filterWords) {
            links = append(links, "https://en.wikipedia.org"+result)
        }
    })

    return links, nil
}
```

buildPathToTarget

```
func buildPathToTarget(target *Node) []string {
    var path []string
    current := target
    for current != nil {
```

```
        path = append([]string{current.Url}, path...)
        current = current.Prev
    }
    return path
}
```

4.3 Frontend

Pada bagian frontend, digunakan Javascript dengan framework React. Bagian ini mengandung beberapa halaman dan komponen yang memilki fungsi masing-masing.

a. **App.js**
   App.js merupakan file yang digunakan untuk mengatur routing pada aplikasi. Routing pada aplikasi ini menggunakan React Router. Library ini memungkinkan untuk membuat single page application pada React tampil sebagai multi page application. Dengan mengatur routing, aplikasi dapat menampilkan halaman lain, yang dalam hal ini dipergunakan untuk menampilkan halaman About dan HowToUse.
b. **Home.js**
   Home.js merupakan file yang berisi komponen Home. Komponen ini berfungsi untuk menampilkan halaman utama dari aplikasi. Pada halaman ini, terdapat judul aplikasi, *header*, serta fungsi utama website yakni menampilkan bagian form input dan hasil pencarian termasuk rute penjelajahan artikel.
c. **Header.js**
   Header.js merupakan file yang berisi komponen Header. Komponen ini berfungsi untuk menampilkan header aplikasi. Pada header ini, terdapat judul aplikasi, serta menu navigasi yang berisi tautan ke halaman utama, About, dan HowToUse. Pada menu navigasi, terdapat fitur highlight yang menandakan halaman yang sedang aktif.
d. **FormInput.js**
   FormInput.js merupakan file yang berisi komponen FormInput. Komponen ini merupakan komponen utama untuk mendapatkan input dari pengguna yang digunakan untuk mencari rute penjelajahan artikel. Form ini terdiri dari input field untuk artikel awal dan akhir, algoritma yang digunakan, serta tombol untuk memulai pencarian. Pada form ini, terdapat fitur saran (autocomplete) artikel yang muncul ketika pengguna mengetikkan artikel pada input field. Terdapat pula fitur swap artikel yang memungkinkan pengguna untuk menukar artikel awal dan akhir dengan sekali tekan.
e. **Result.js**
   Result.js merupakan file yang berisi komponen Result untuk menampilkan hasil pencarian rute penjelajahan artikel. Komponen ini hanya akan muncul ketika proses pencarian telah selesai. Pada komponen ini, terdapat informasi artikel yang telah

dicek, jumlah rute yang ditemukan, artikel yang diklik, waktu eksekusi, serta visualisasi rute penjelajahan artikel dalam bentuk graf.

f. **GraphVisualization.js**
GraphVisualization.js merupakan file yang berisi komponen GraphVisualization. Komponen ini berfungsi untuk menampilkan visualisasi graf rute penjelajahan artikel. Terdapat konversi data rute penjelajahan artikel ke dalam bentuk graf yang dapat ditampilkan pada halaman. Pada graf ini, pengguna dapat memperbesar, memperkecil, menarik, dan memindahkan node pada graf. Pengguna juga dapat mengklik node untuk membuka artikel pada tab baru. Semua fitur ini diatur di dalam komponen GraphVisualization.

g. **HowToUse.js**
HowToUse.js merupakan file yang berisi informasi mengenai panduan penggunaan aplikasi. Komponen ini berisi langkah-langkah penggunaan mulai dari pengisian form input hingga menampilkan hasil pencarian rute penjelajahan artikel.

h. **About.js**
About.js merupakan file yang berisi informasi mengenai aplikasi. Komponen ini berisi deskripsi singkat mengenai aplikasi web, algoritma yang digunakan, teknologi yang digunakan, serta identitas tim pengembang aplikasi.

4.4 Tata Cara Penggunaan Program

Program dapat dijalankan dengan menggunakan docker. Untuk menjalankan program menggunakan docker, pengguna dapat terlebih dahulu membuka docker desktop pada device. Setelah itu, buka terminal, masuk ke direktori src dari direktori utama (Tubes2_Jebir/src). Kemudian, jalankan perintah `docker-compose up` pada terminal untuk menjalankan program. Program backend akan berjalan pada port 8000, sedangkan program frontend akan berjalan pada port 3000. Website akhir dapat diakses melalui browser dengan membuka alamat http://localhost:3000. Setelah program berhasil berjalan, pengguna dapat langsung menggunakan program dengan:

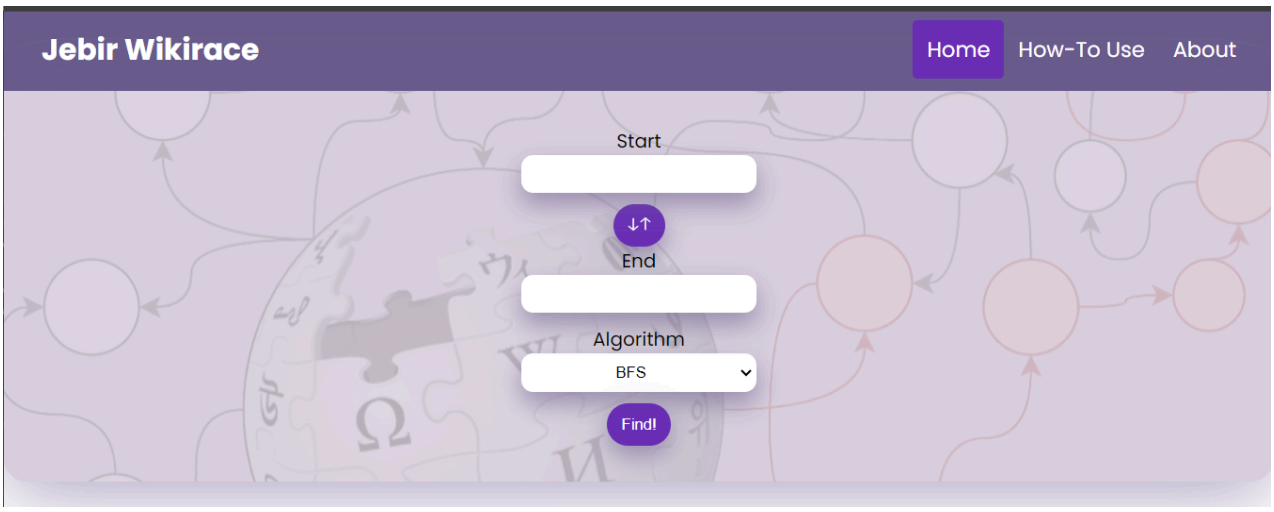
1. Masukkan judul artikel awal pada kolom input 'Start'.
2. Masukkan judul artikel tujuan pada kolom input 'End'.
3. Pilih algoritma yang ingin digunakan pada dropdown 'Algorithm'. Pada bagian ini terdapat dua algoritma utama yaitu algoritma BFS dan IDS, serta dua algoritma tambahan yaitu BFS All Path dan IDS All Path sebagai opsi untuk menampilkan semua rute yang mungkin ditemukan.
4. Klik tombol 'Find!' untuk memulai pencarian rute penjelajahan menuju artikel tujuan.
5. Tunggu hingga program menemukan rute terpendek dan menampilkan hasilnya yaitu rute terpendek yang ditemukan, jumlah artikel yang telah dicek, jumlah artikel yang diklik, dan waktu eksekusi yang dibutuhkan untuk menemukan rute tersebut.
6. Pada bagian hasil ini, terdapat visualisasi graf yang menghubungkan artikel Wikipedia yang telah dicari rute terpendeknya dengan artikel-artikel yang terhubung dengan artikel

tersebuKlik simpul artikel pada graf untuk membuka artikel wikipedia tersebut pada tab baru.

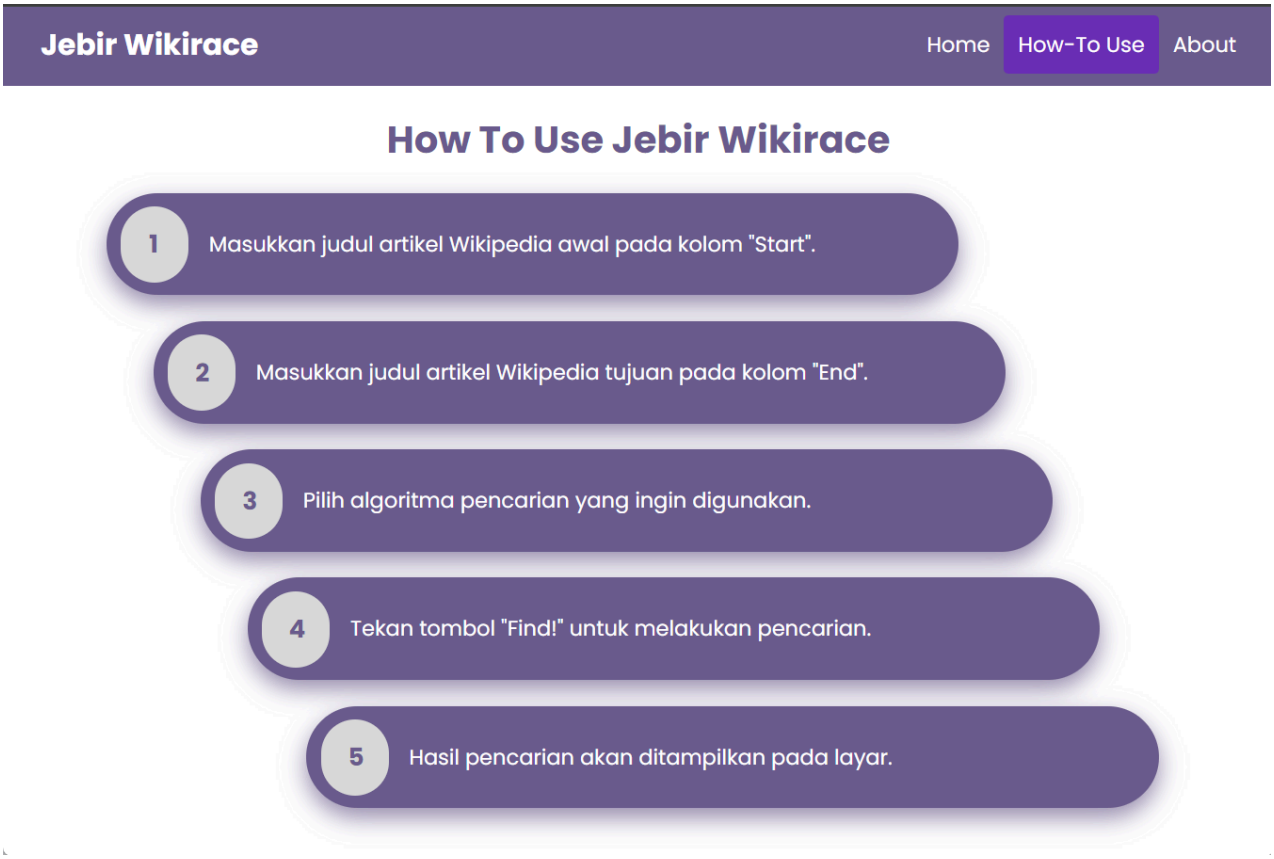
7. Ulangi dari langkah 1 untuk melakukan pencarian lainnya.
8. Pada halaman awal, terdapat tombol How To Use yang dapat diakses untuk membuka panduan penggunaan program web application ini.
9. Terdapat pula tombol About yang dapat diakses untuk melihat informasi mengenai program web application ini beserta identitas pengembang.

4.5 Hasil Pengujian

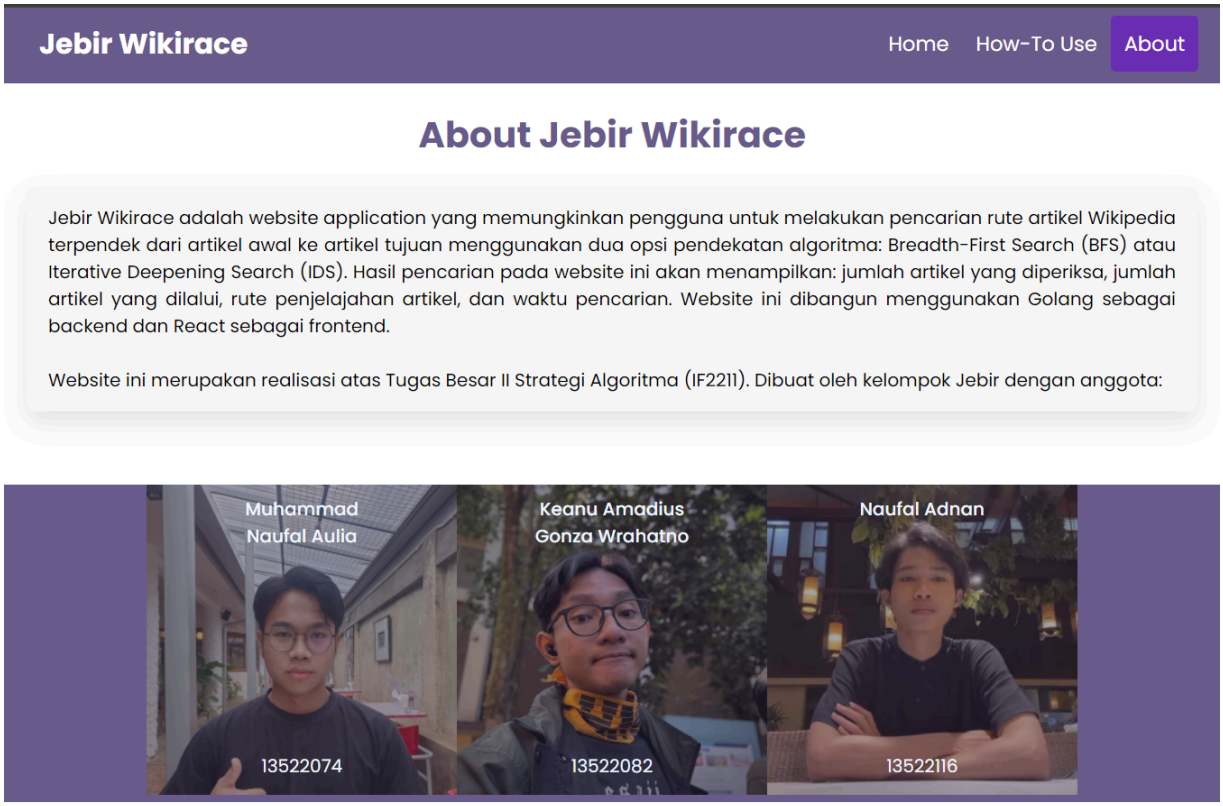
4.5.1 Antarmuka Website

Tampilan Awal Halaman Home


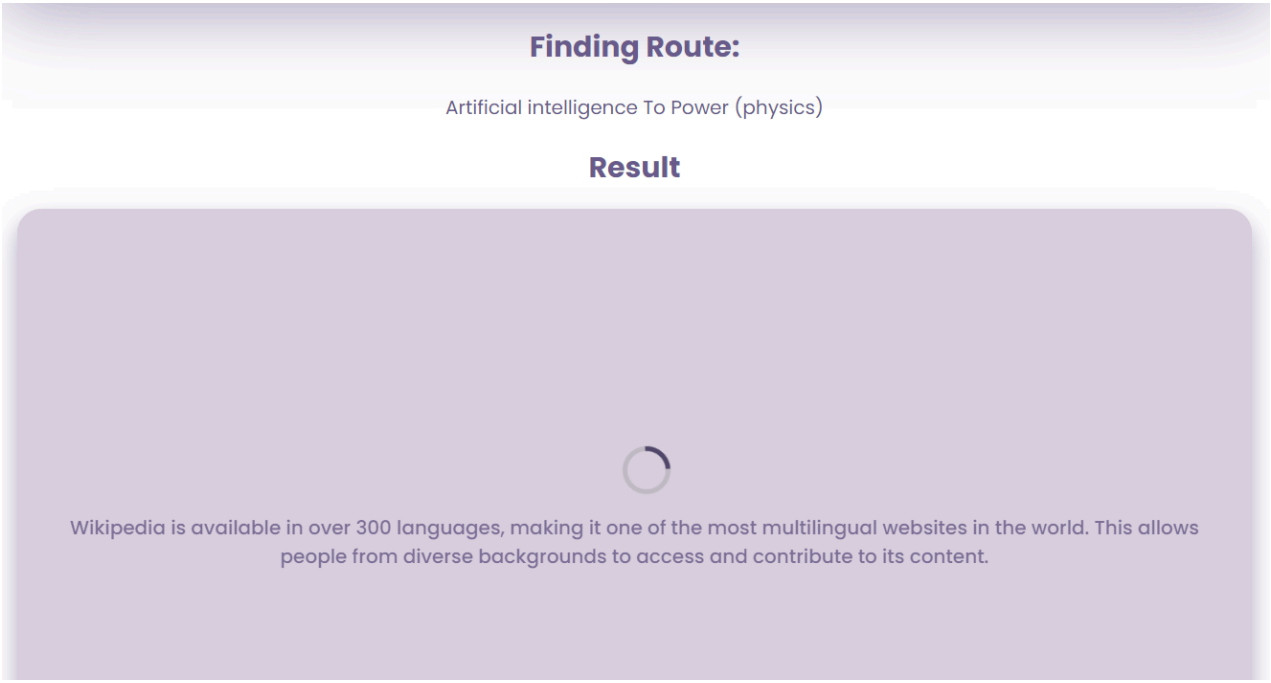
Halaman How-To Use



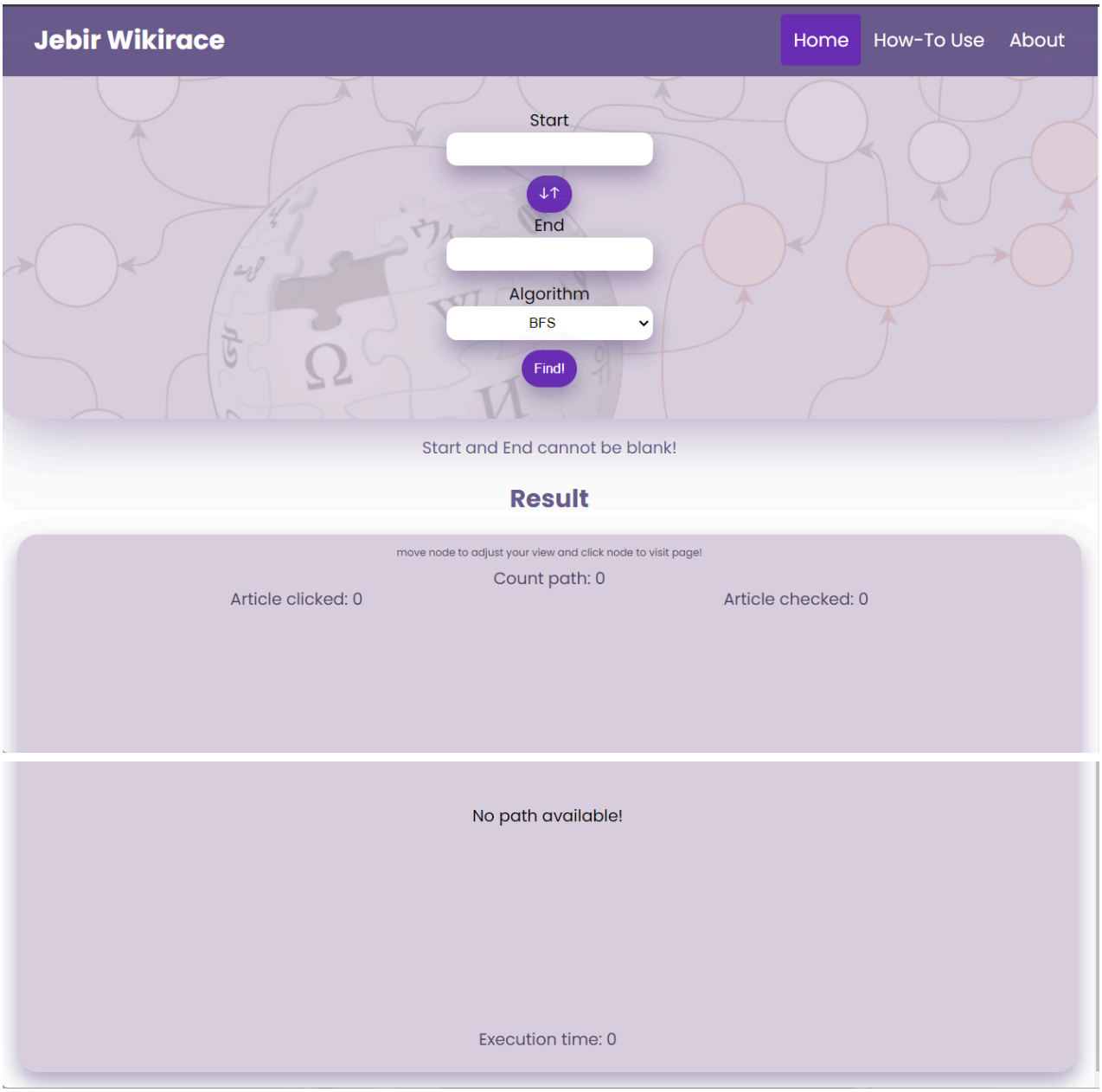
Halaman About



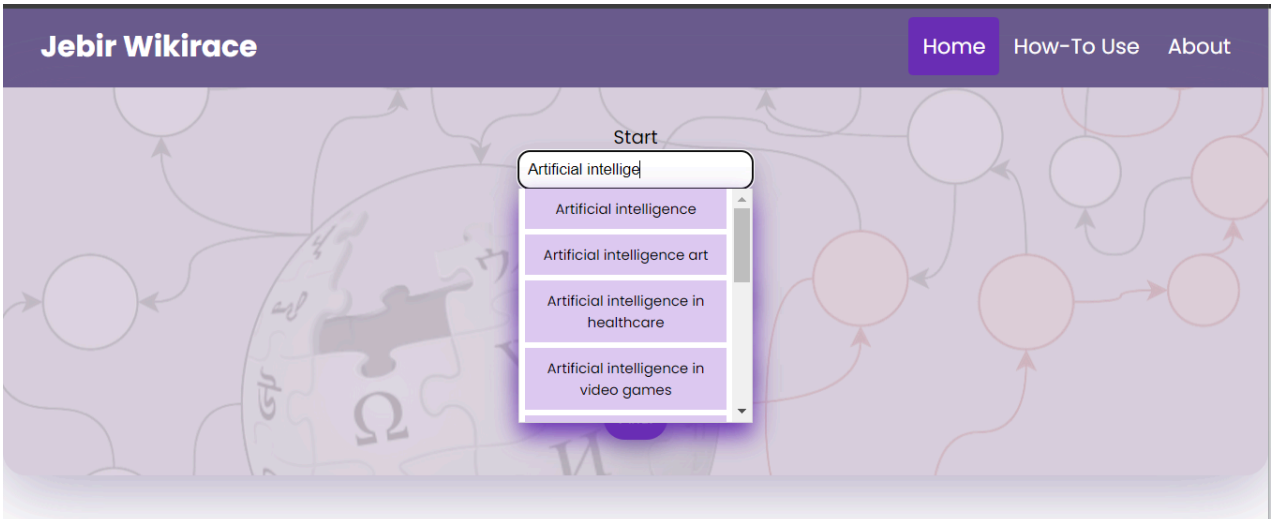
Loading



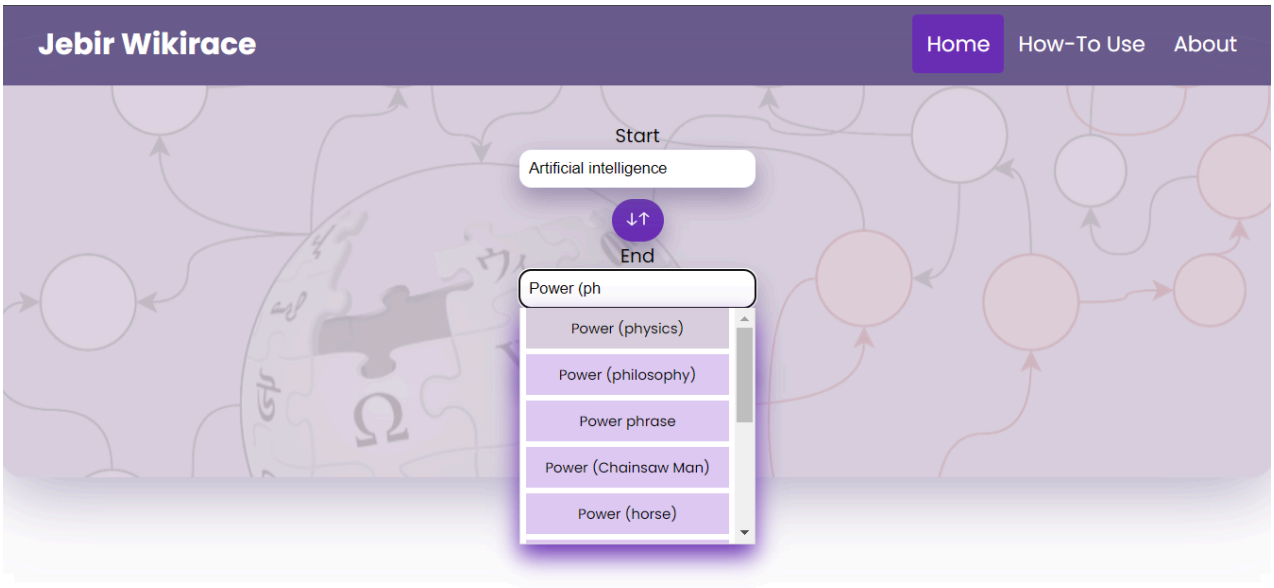
4.5.2 Field Input

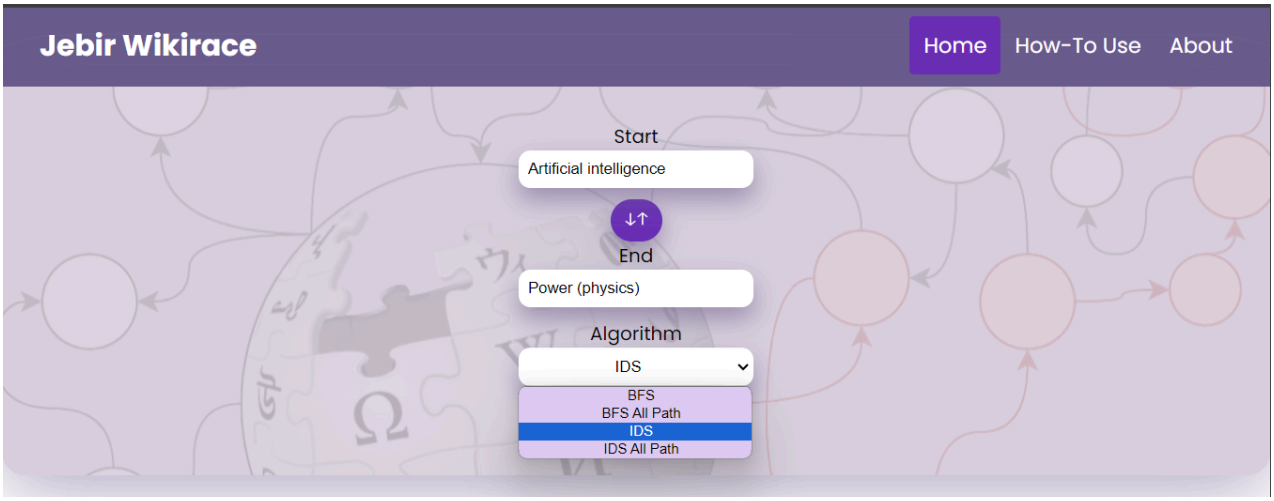
Input Ketika Start dan End Kosong


Suggestion Input Start

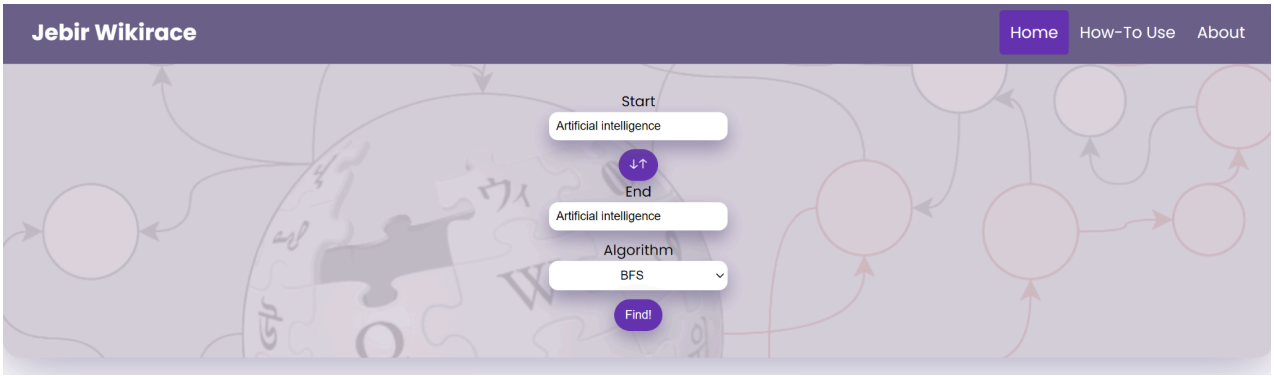


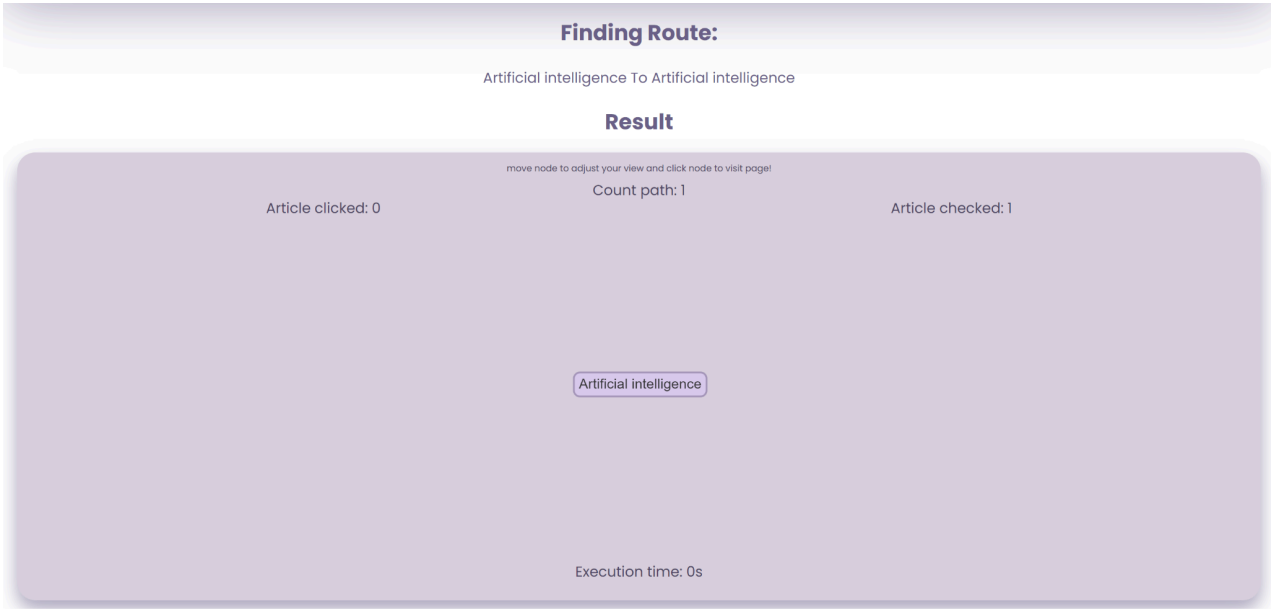
Suggestion Input End



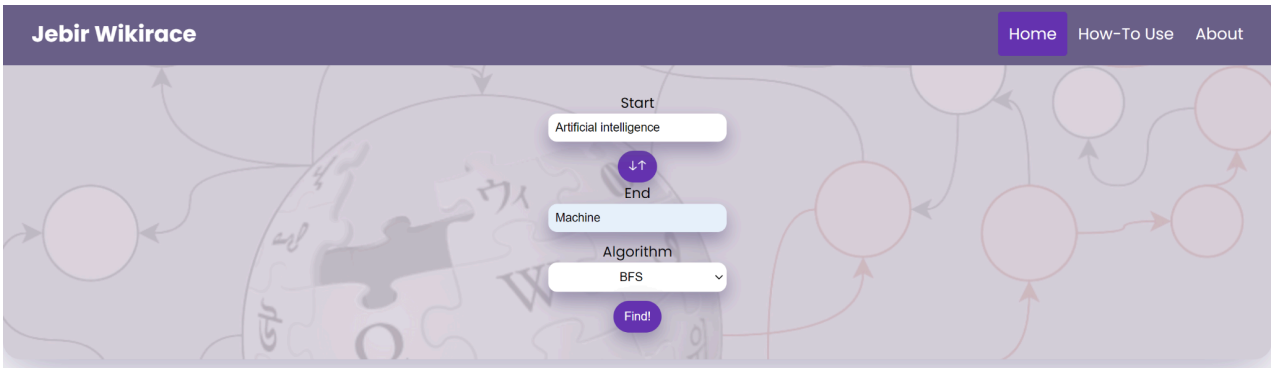
Dropdown Input Algoritma


4.5.3 Uji Coba Kasus

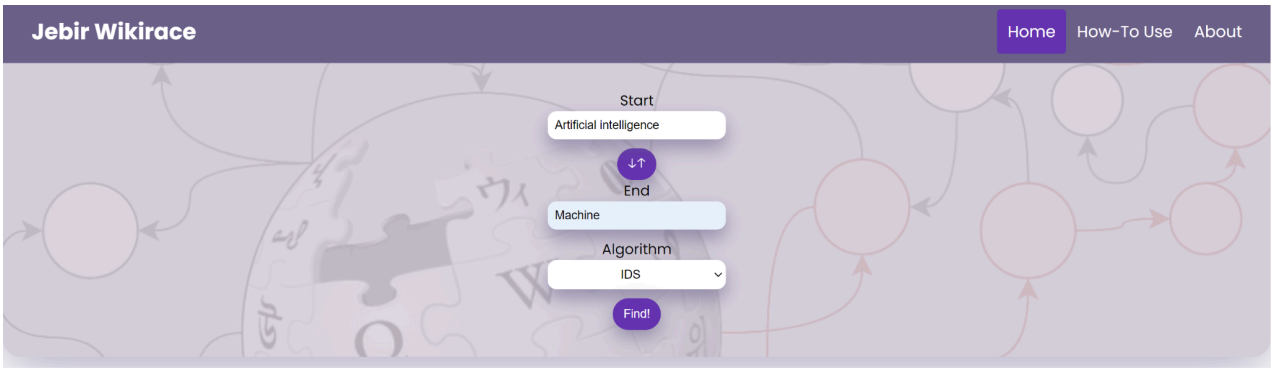
Pencarian dari Start ke End yang Sama


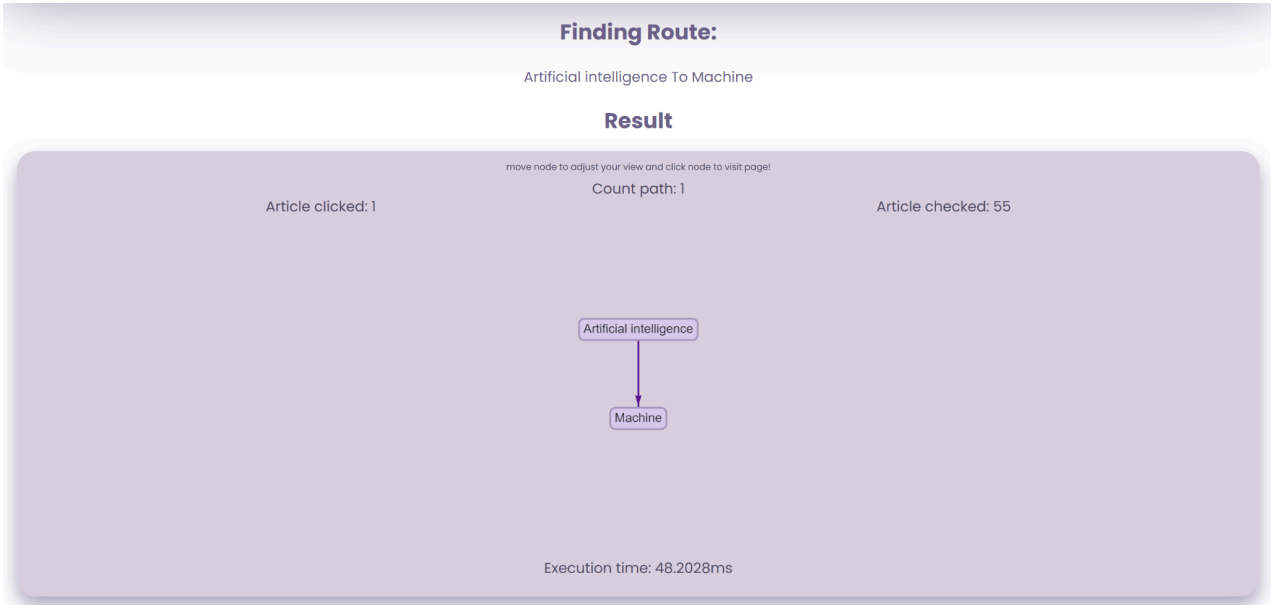


Pencarian dari Artificial intelligence ke Machine [BFS]

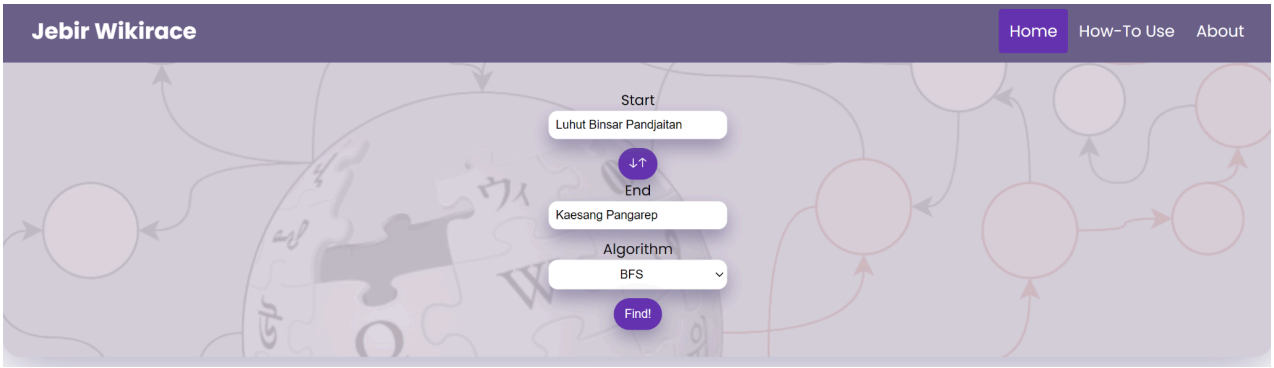


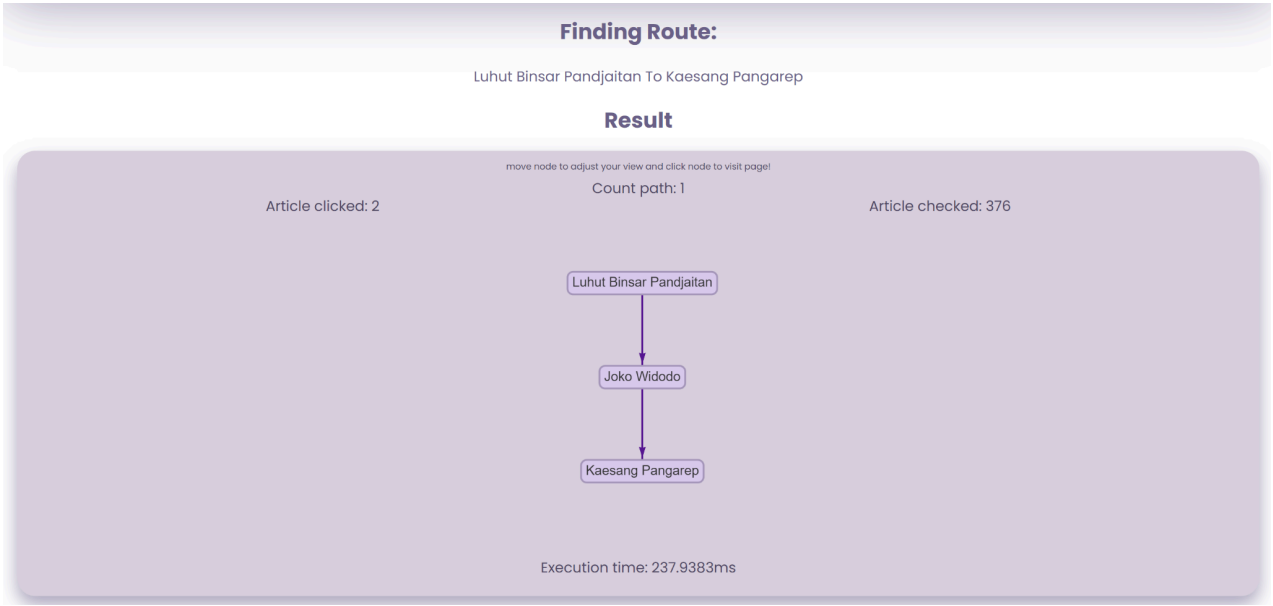
Pencarian dari Artificial intelligence ke Machine [IDS]



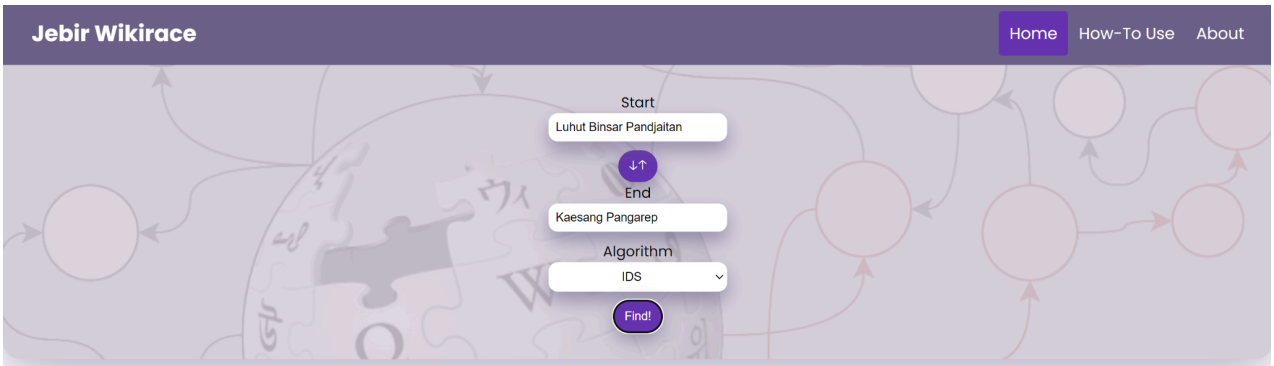


Pencarian dari Artificial Luhut Binsar Panjaitan ke Kaesang Pangarep [BFS]

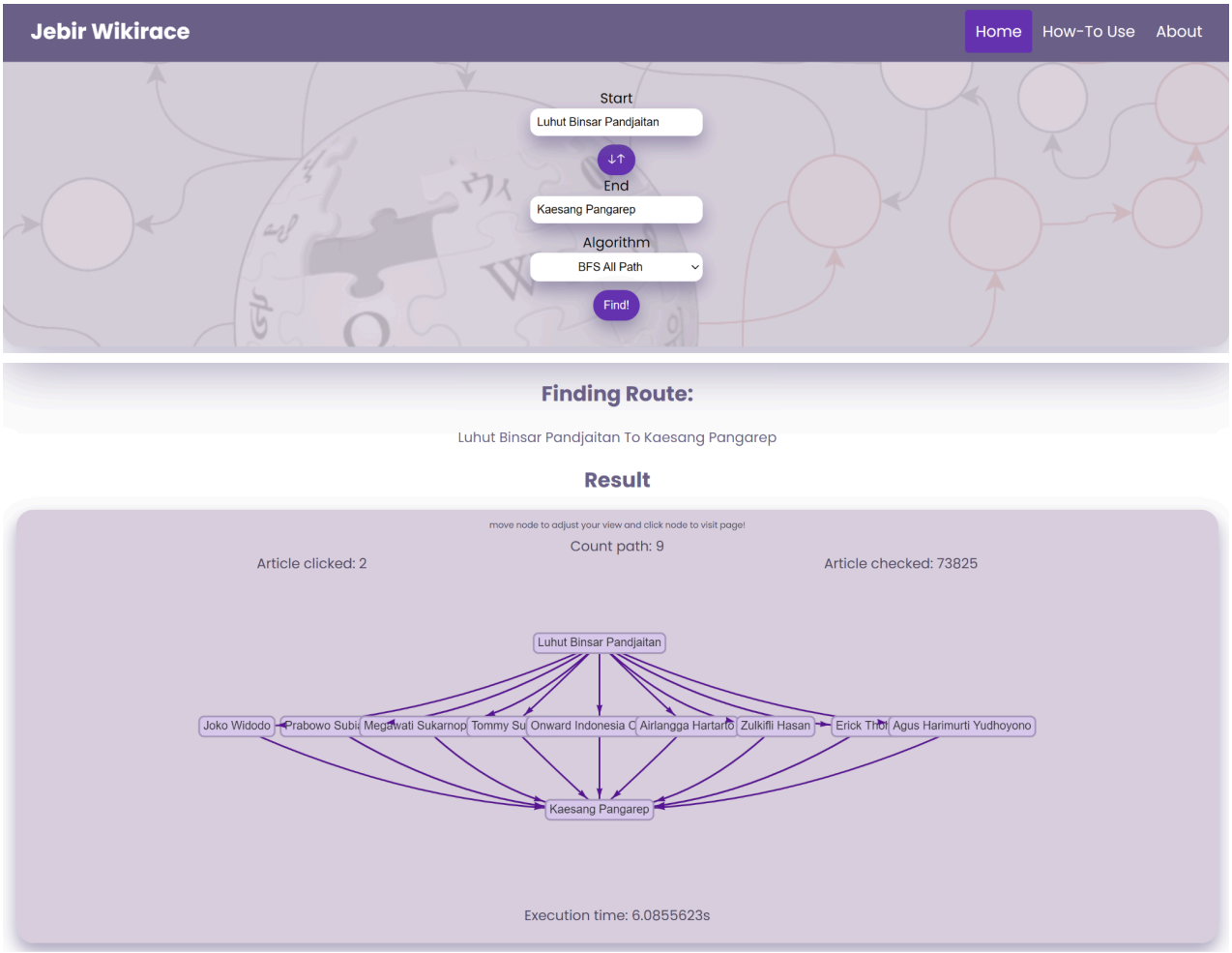




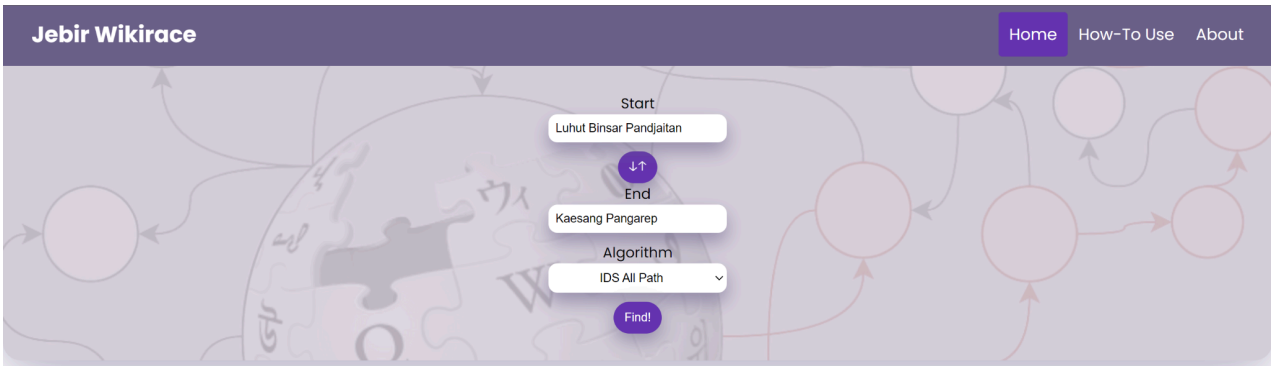
Pencarian dari Luhut Binsar Panjaitan ke Kaesang Pangarep [IDS]

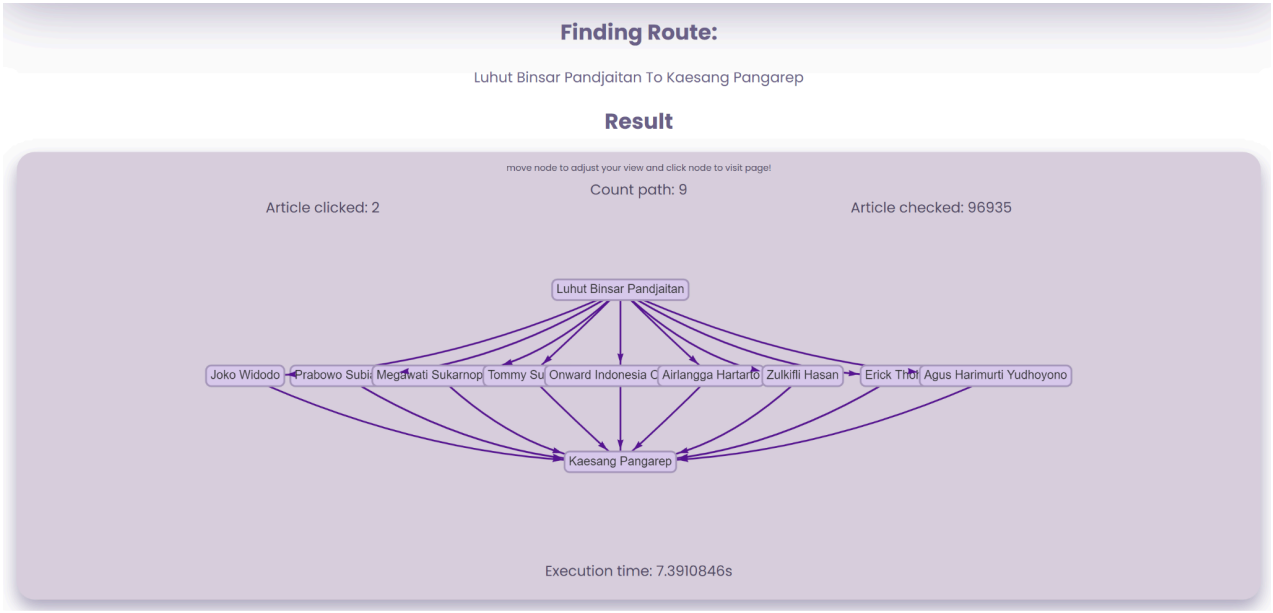


Pencarian dari Luhut Binsar Panjaitan ke Kaesang Pangarep [BFS All Path]

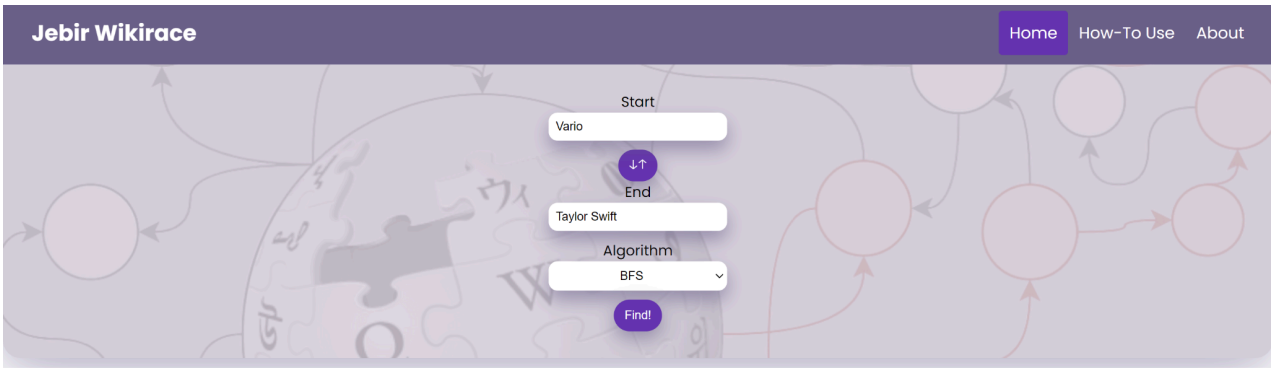


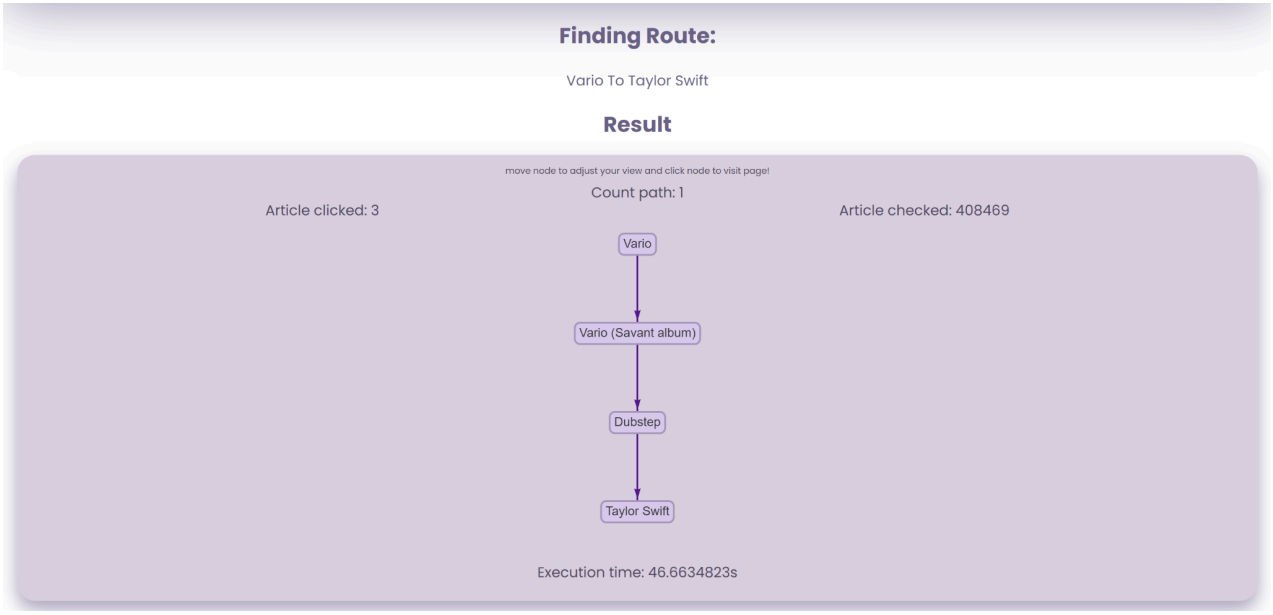
Pencarian dari Luhut Binsar Panjaitan ke Kaesang Pangarep [IDS All Path]



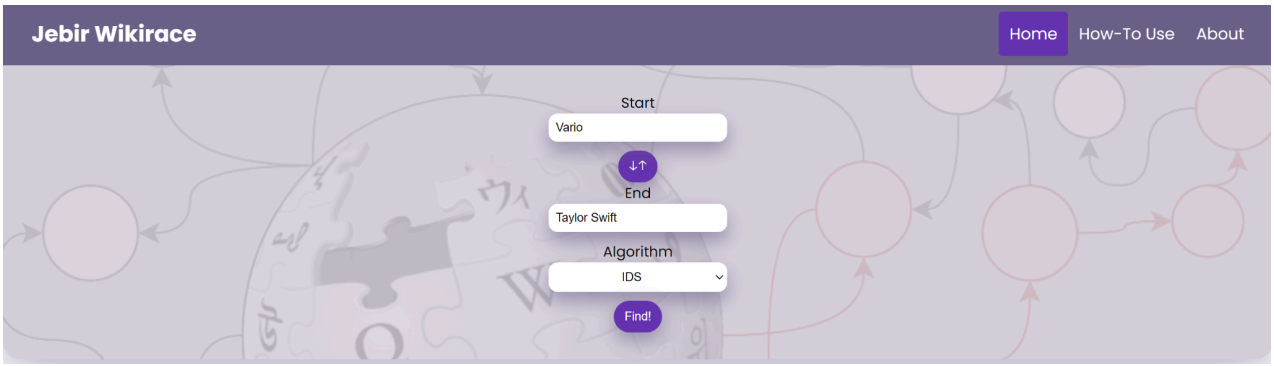


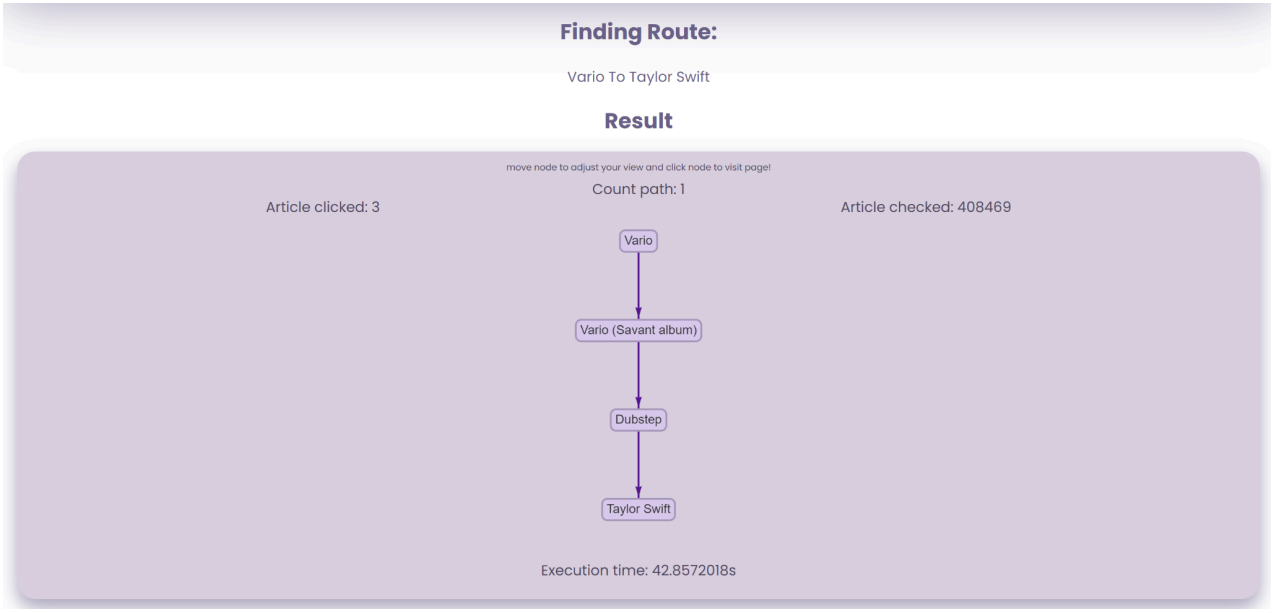
Pencarian dari Vario ke Taylor Swift [BFS]



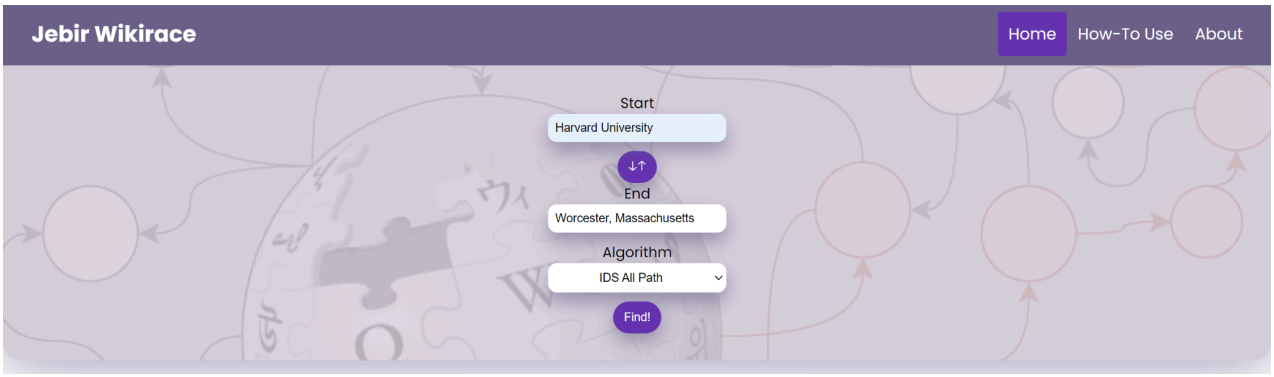


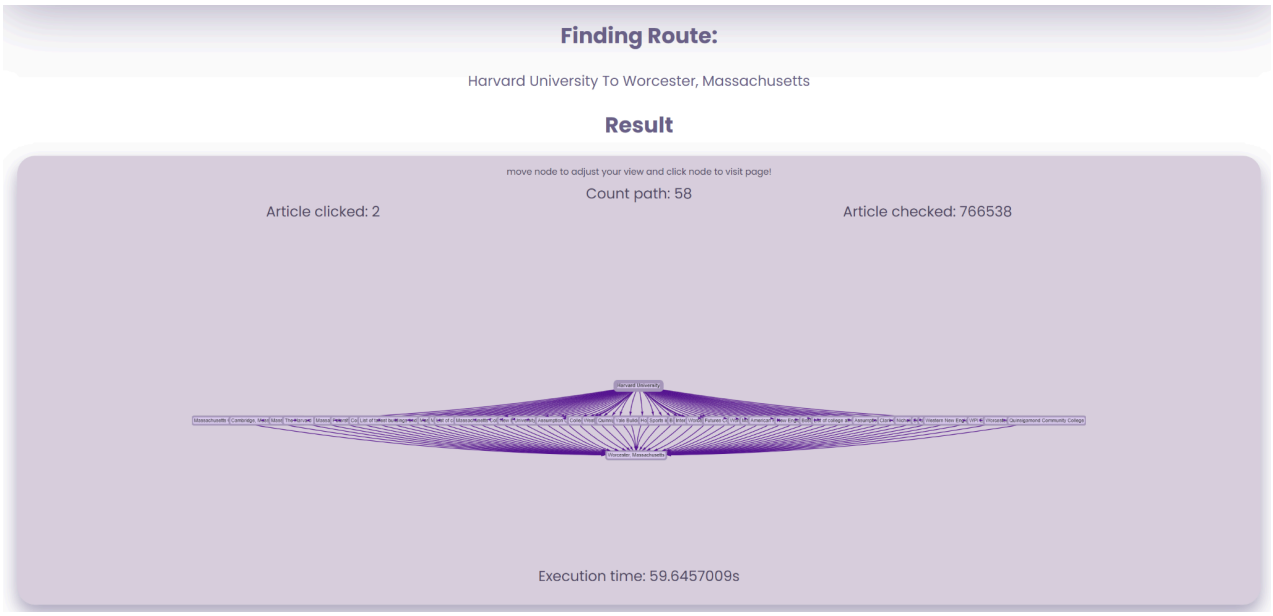
Pencarian dari Vario ke Taylor Swift [IDS]



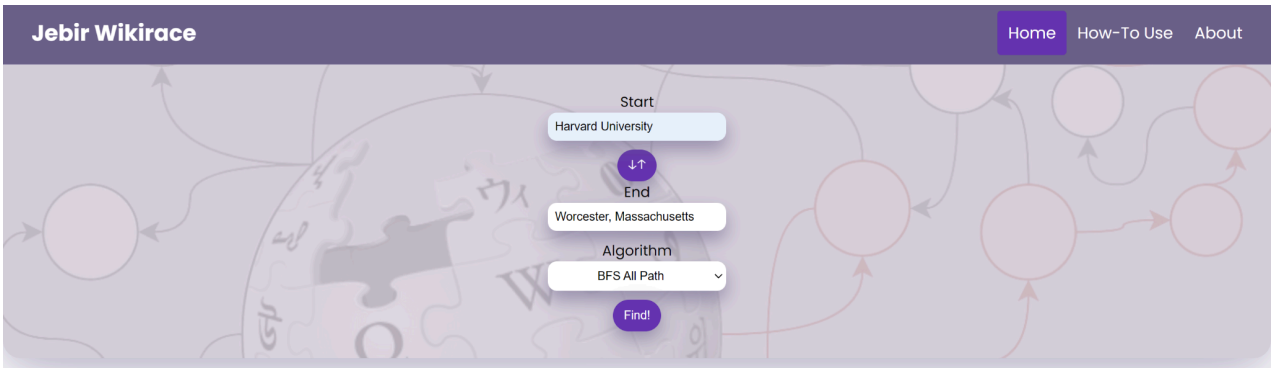


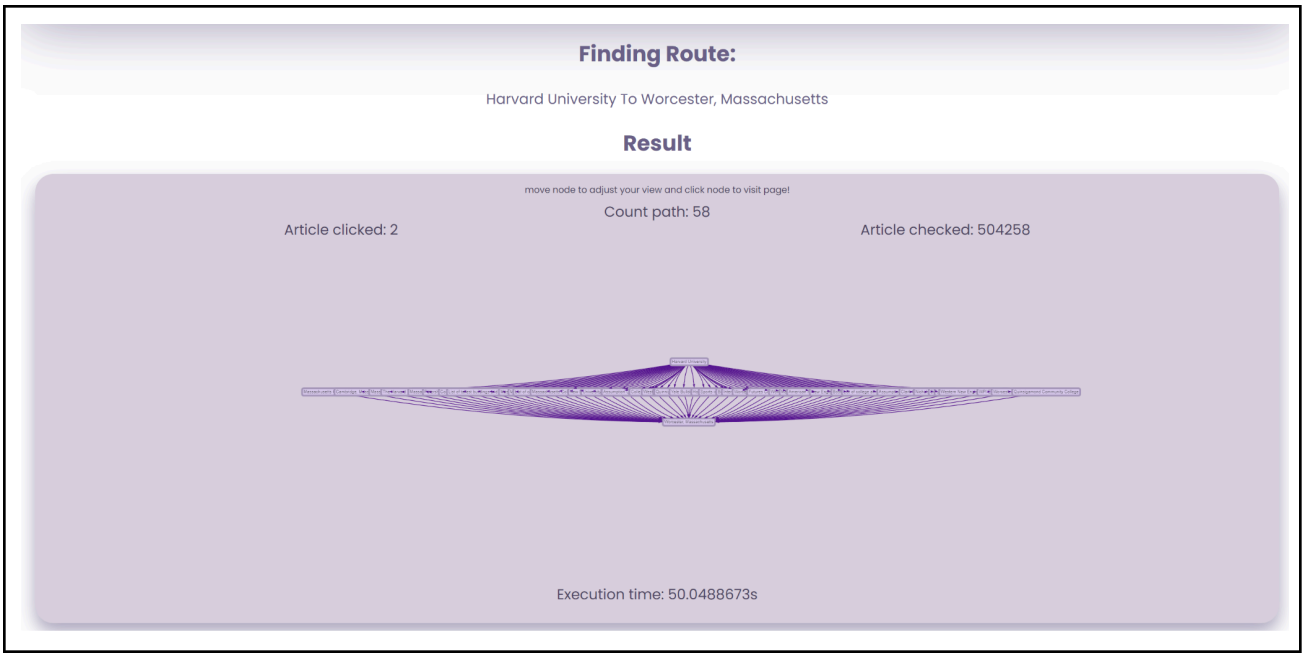
Pencarian dari Harvard University ke Worcester, Massachusetts [IDS All Path]





Pencarian dari Harvard University ke Worcester, Massachusetts [BFS All Path]





4.6 Analisis Hasil Pengujian

Penggunaan antara metode BFS dan IDS menggunakan pendekatan yang berbeda. Namun keduanya sama sama digunakan untuk mencari jalur terpendek. Efisiensi kedua algoritma ini tergantung dari kasus persoalannya. Berikut merupakan hasil analisis terhadap pengujian yang telah dilakukan.

Dari semua pengujian single path baik menggunakan IDS ataupun BFS, didapatkan bahwa semuanya memiliki jumlah panjang path yang sama dan jumlah artikel diperiksa yang sama. Ini menandakan bahwa baik IDS maupun BFS sama sama berusaha untuk menyelesaikan terlebih dahulu pencarian di suatu kedalaman untuk memungkinkan path terpendek. Yang membedakan mereka yaitu proses atau urutan pencariannya. Dimana IDS perlu iterasi untuk tiap kedalaman dan tiap iterasi itu menerapkan konsep DLS yang melakukan pencarian berdasarkan kedalaman terlebih dahulu. Proses pencarian BFS tidak menggunakan iterasi namun menyelesaikan eksplorasi di suatu kedalaman lalu baru pindah ke kedalaman berikutnya.

Perbedaan pengujian antara BFS dan IDS dapat dilihat dari waktu eksekusi programnya. Waktu pada algoritma IDS memakan waktu sedikit lebih lama karena algoritma IDS berlangsung dengan iterasi dimana artikel yang telah dicek akan dicek kembali pada iterasi berikutnya. Berbeda dengan BFS yang prosesnya tidak menggunakan iterasi melainkan menggunakan queue sampai ditemukan link tujuan.

Kami juga melakukan pengujian untuk mencari multiple path. Didapatkan data bahwa pencarian multiple path ini membutuhkan waktu yang lebih lama dibandingkan dengan pencarian single path. Hal ini dikarenakan jika algoritma DFS atau IDS sudah menemukan link tujuan,

program tidak akan berhenti disitu. Program akan terus berlanjut sampai semua simpul pada kedalaman tertentu telah dikunjungi. Ini diperlukan untuk menemukan semua path dan menampilkannya ke layar.

Pada pengujian tersebut, kami juga mendapatkan kesimpulan bahwa semakin dalam link target itu, maka semakin banyak pula artikel yang diperiksa dan semakin lama pula waktu eksekusi programnya.

BAB V
PENUTUP

5.1 Kesimpulan

Dalam tubes ini, digunakan algoritma IDS bukan DFS karena IDS dapat menemukan rute terpendek walaupun pendekatannya menurut kedalamannya terlebih dahulu. Hal ini dikarenakan IDS prosesnya iteratif membesar menurut kedalamannya. Sehingga memaksa penggunaan DFS dengan menerapkan limit untuk mengecek keseluruhan simpul di suatu kedalaman terlebih dahulu. Jika semua simpul sampai kedalaman tertentu belum ditemukan target, maka akan melanjutkan pencarian ke kedalaman yang berikutnya. Hal ini hampir mirip dengan BFS yang sama sama mencari semua simpulnya terlebih dahulu di suatu kedalaman baru melanjutkan ke kedalaman berikutnya. Bedanya adalah cara atau urutan pencarian simpul simpulnya.

Kami juga berhasil mengimplementasikan sebuah web application yang memanfaatkan algoritma Breadth-First Search (BFS) dan Iterative Deepening Search (IDS) untuk mencari rute penjelajahan artikel dari artikel Wikipedia awal ke artikel Wikipedia tujuan. Penggunaan algoritma-algoritma tersebut telah membantu dalam menemukan rute penjelajahan dengan efisien dan efektif dengan kelebihan dan kekurangannya masing-masing.

Pengembangan aplikasi dilakukan dengan menggunakan bahasa pemrograman Go untuk bagian backend dan React JS untuk bagian frontend. Kombinasi kedua teknologi ini memungkinkan kami untuk menciptakan sebuah web application yang responsif dan mudah digunakan. Melalui tugas ini, kami juga banyak belajar penggunaan framework dan teknologi website terkini.

5.2 Saran

Berdasarkan hasil pengembangan aplikasi ini, kami memiliki beberapa saran untuk perbaikan dan pengembangan di masa yang akan datang:

1. Optimisasi Algoritma:

   Algoritma BFS dan IDS dapat dioptimalkan lebih lanjut dengan teknik-teknik yang mempercepat proses pencarian dan mengurangi penggunaan sumber daya. Menggunakan pendekatan heuristik atau memanfaatkan teknik paralelisasi bisa menjadi area eksplorasi yang baik untuk iterasi berikutnya.

2. Peningkatan Efisiensi Backend:

   Meskipun aplikasi ini telah berfungsi dengan baik, ada peluang untuk meningkatkan efisiensi backend dengan penggunaan teknologi caching atau optimalisasi akses database. Implementasi caching pada sisi server dapat mempercepat waktu respon dan mengurangi beban server.

5.3 Refleksi

Dalam proses pengembangan aplikasi website ini, kami belajar banyak tentang strategi algoritma khususnya pada pendekatan Breadth-First Search (BFS) dan Iterative Deepening Search (IDS), penggunaan bahasa pemrograman Go, dan pengembangan antarmuka pengguna menggunakan React JS. Kami juga memperoleh pemahaman yang lebih mendalam tentang pentingnya merancang algoritma pencarian yang efisien dalam menyelesaikan permasalahan penelusuran informasi. Di sini, kami juga belajar bagaimana untuk mengintegrasi kedua bagian website ini.

Selain itu, kolaborasi antaranggota kelompok dalam pengembangan aplikasi ini juga memberikan pengalaman berharga dalam koordinasi dan integrasi antara kedua bagian sistem. Kami berharap bahwa aplikasi yang kami buat dapat bermanfaat bagi kami sendiri dan bagi orang lain dalam menjelajahi informasi di Wikipedia, bahan pembelajaran mengenai algoritma, serta menjadi landasan untuk pengembangan lebih lanjut di masa yang akan datang.

LAMPIRAN

Link repository : https://github.com/nanthedom/Tubes2_Jebir
Link video : https://youtu.be/7pQbOzHwpT8

DAFTAR PUSTAKA

Munir, Rinaldi. 2024. Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1) (Versi update 2024). Diakses dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf

Munir, Rinaldi. 2024. Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 2). Diakses dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf