

**Laporan Tugas Besar 2  
IF2211 Strategi Algoritma  
WikiRace Solver  
Semester II Tahun 2023/2024**



Disusun Oleh:  
Kristo Anugrah 13522024  
Farhan Nafis Rayhan 13522037  
Adril Putra Merin 13522068

**Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung  
2024**

# BAB 1

## Deskripsi Tugas

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



Gambar 1.1 Ilustrasi Graf permainan WikiRace

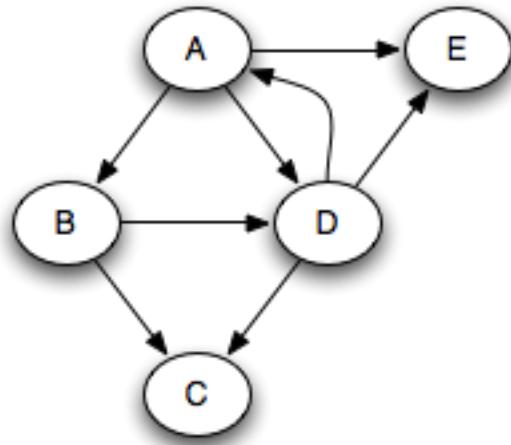
Dalam tugas ini, kami membuat sebuah program berbasis web menggunakan bahasa pemrograman Go yang dapat menyelesaikan permainan WikiRace. Program ini mengimplementasikan algoritma *Breadth First Search (BFS)* dan *Iterative Deepening Search (IDS)* yang telah dipelajari selama perkuliahan IF2211 Strategi Algoritma. Program ini menerima masukkan judul artikel awal, judul artikel akhir, serta algoritma yang digunakan. Kemudian, program akan menampilkan jumlah artikel yang diperiksa, waktu total pencarian, serta visualisasi seluruh rute penjelajahan artikel. Program ini bersifat lokal, sehingga tidak di-deploy kepada internet.

## BAB 2

# Landasan Teori

### 2.1. Penjelajahan Graf

Pada penyelesaian permainan WikiRace ini, perlu dilakukan *traversal* (penjelajahan) graf. Penjelajahan graf secara umum berarti mengunjungi simpul-simpul yang terdapat di dalam graf secara sistematis untuk mencari solusi dari permasalahan yang direpresentasikan oleh graf. Terdapat 2 jenis algoritma pencarian pada graf, yaitu pencarian secara tanpa informasi (*uninformed/blind*) dan pencarian dengan informasi (*informed Search*). Pada kasus ini, diperlukan algoritma yang dapat melakukan pencarian tanpa informasi karena tidak adanya pendekatan *heuristik* mengenai *state* apakah yang lebih baik dibandingkan yang lainnya. Terdapat 2 buah algoritma umum yang termasuk dalam jenis tanpa informasi, yang akan digunakan dalam program ini. Yaitu algoritma *Breadth First Search (BFS)* dan *Iterative Deepening Search (IDS)*.

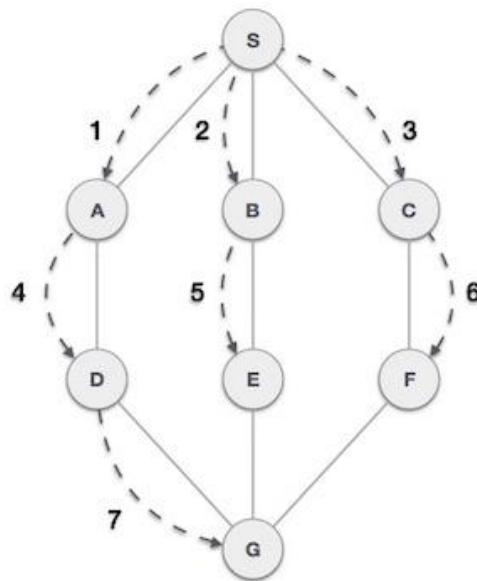


Gambar 2.1.1 Ilustrasi Penjelajahan (Traversal) Graf

### 2.2. Breadth First Search

Pencarian dengan algoritma *Breadth First*, atau BFS adalah salah satu algoritma yang paling banyak digunakan untuk *traversal* graf. Algoritma ini ditemukan oleh Konrad Zuse pada tahun 1945, lalu ditemukan kembali pada tahun 1959 oleh Edward F. Moore dalam implementasi untuk mencari jalur tercepat pada sebuah labirin. Algoritma ini utamanya dicirikan dengan pencarian melebar.

Penjelajahan dimulai dari suatu simpul akar dengan level 0. Awalnya, kita akan mengunjungi setiap simpul yang bertetangga dengan simpul akar, ini merupakan pencarian level 1. Secara umum, pada setiap level selanjutnya, kita akan melebarkan pencarian dengan mengunjungi tetangga dari simpul yang kita kunjungi pada level sebelumnya secara satu-persatu. Amati bahwa kita tidak perlu mengunjungi kembali simpul yang telah dikunjungi sebelumnya, karena hal ini akan membuat algoritma BFS tidak efektif. Pencarian per level terus dilaksanakan sampai simpul tujuan ditemukan, atau seluruh simpul dalam graf sudah dijelajahi.



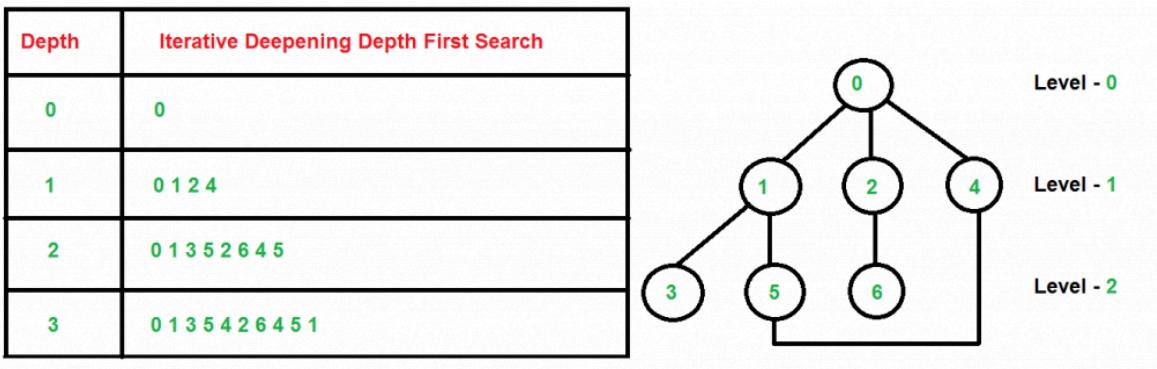
Gambar 2.2.1 Contoh pencarian BFS pada graf

### 2.3. Iterative Deepening Search

Sering pula disebut dengan iterative deepening depth-first search, pencarian ini merupakan salah satu variasi dari *Depth First Search (DFS)*. Sejatinya, DFS adalah algoritma dengan pendekatan mendalam suatu jalur. Dimulai dari simpul akar, jalur ditelusuri sampai kita bertemu ujung, lalu kita melanjutkan pencarian pada jalur selanjutnya. Hal ini dilakukan sampai semua jalur telah ditelusuri atau ditemukan suatu jalinan yang ingin dicari.

Pada permainan WikiRace ini, sulit untuk dilakukan DFS seperti umumnya karena banyaknya halaman pada Wikipedia mengakibatkan ujung dari jalur sangat sulit untuk dicapai. Akibatnya, diperlukan IDS, yaitu DFS yang telah dibatasi jarak pencariannya. Pada awalnya, *cut-off* (jarak) pencarian adalah 1 sehingga dilakukan serangkaian DFS dengan jarak 1 sampai semua jalur dilewati. Untuk setiap langkah

selanjutnya, jarak dinaikkan sehingga pencarian DFS akan lebih dalam pada setiap langkahnya. Jarak akan selalu ditambah 1 sampai ditemukan simpul yang diinginkan.



**Gambar 2.3.1** Contoh pencarian IDS pada graf

## 2.4. Web Based Solver using Go

Go merupakan bahasa pemrograman *open source* yang awal mulanya dikembangkan oleh sebuah tim dari *Google* pada 2007. Bahasa ini banyak digunakan oleh berbagai perusahaan besar, termasuk *Google*, untuk mengembangkan produk dalam skala masif. Go sangat populer akan performanya yang cepat namun tetap aman, serta kemampuannya dalam membuat aplikasi berbasis web yang *scalable*.

Go telah menyediakan berbagai *standard library* yang teruji baik performanya. Salah satunya adalah sistem konkuren (concurrency) yang sudah disediakan oleh bahasa ini (*built-in*) serta *package* bernama “sync”. Dalam aplikasi berbasis web ini, *package sync* banyak digunakan karena hal ini memberikan optimasi yang cukup besar melalui *multithreading* dan *concurrent process*.

Selain *package sync*, “net/http” turut serta digunakan dalam tugas ini. Package ini memberikan implementasi untuk *server HTTP* secara praktis. Hal ini digunakan untuk mengembangkan *Application Programming Interface (API)* yang akan mendukung *Backend/Server Side* dari aplikasi web.

Sedangkan untuk *Frontend*, atau sisi dari *client*, digunakan bahasa pemrograman web yaitu *TypeScript*. Bahasa ini merupakan bahasa yang mirip dengan *JavaScript*, tetapi bahasa ini memiliki sintaks tambahan yang membuatnya *strongly-typed*. *TypeScript* banyak digunakan karena ia memberikan peringatan pada kesalahan kode lebih dahulu sehingga menjamin keamanan kode pada skala besar sekalipun.

*TypeScript* memiliki framework bernama *Next.js* yang merupakan salah satu *framework* untuk menciptakan antarmuka yang elegan dan modern. *Framework* ini sangat efektif karena memiliki fitur untuk menciptakan *styling* antarmuka, *fetching* data dari API yang sudah disimplifikasi, serta integrasi dengan bahasa *TypeScript*. *Next.js*

memiliki keunggulan dalam antarmuka yang dinamis juga interaktif, akibatnya framework ini kami gunakan dalam pembuatan aplikasi web ini.

## BAB 3

# Analisis Pemecahan Masalah

### 3.1. Langkah Pemecahan Masalah

#### 3.1.1. Algoritma Iterative Deepening Search

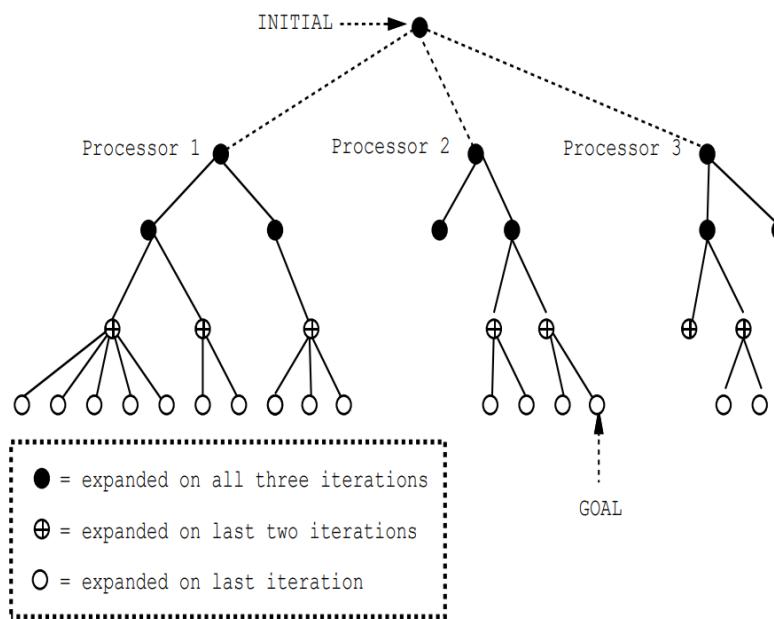
Algoritma IDS membutuhkan dua input string, yaitu alamat simpul mulai S dan alamat simpul akhir E. Pada algoritma ini, seluruh simpul tetangga dari simpul mulai akan ditelusuri dengan teknik *depth-limited search*. *Depth-limited search* adalah sebuah variasi dari teknik *depth-first search*, dimana program akan berhenti menelusuri simpul tetangga jika kedalaman saat itu melebihi batas kedalaman yang telah ditentukan.

Program akan mencoba batas kedalaman 1 untuk memulai. Jika simpul akhir E telah ditemukan dalam proses tersebut, maka program akan mengembalikan hasil kedalaman 1 dengan jalur yang telah ditemukan. Namun, apabila simpul akhir masih belum ditemukan, batas kedalaman akan ditambah menjadi 2. Batas kedalaman akan terus menerus ditambah selama simpul akhir masih belum dapat diraih. Batas kedalaman maksimum yang diterapkan adalah 10. Jika program masih tidak dapat menemukan simpul akhir hingga kedalaman 10, diasumsikan tidak ada jalur yang memenuhi. Batasan 10 diterapkan karena sifat sisi dari artikel wikipedia yang terhubung erat. Kelompok kami mengobservasi bahwa banyak simpul yang dikunjungi bertumbuh secara eksponensial di setiap kedalaman, dengan kurang lebih penambahan 800 simpul pada setiap kedalaman. Dengan kedalaman 10, kira-kira program dapat mengunjungi  $800^{10}$  artikel wikipedia ( $\approx 10^{29}$  simpul). Namun, diketahui hanya terdapat kurang lebih 6 juta ( $6 \times 10^6$ ). Ini artinya kedalaman 10 sangatlah cukup untuk dijadikan batas, karena banyak simpul yang dapat diraih jauh melebihi banyak simpul yang ada.

Pada persoalan wikirace kali ini, kelompok kami mengobservasi bahwa sebagian besar waktu dihabiskan untuk membangkitkan simpul tetangga, yaitu saat *scraping* sebuah artikel. Untuk mempercepat jalannya program, algoritma IDS dilengkapi dengan *cache*, dimana simpul tetangga tidak perlu dibangkitkan kembali jika sudah dibangkitkan sebelumnya. Simpul tetangga yang telah pernah dibangkitkan hanya perlu diambil dari *cache*. Untuk simpul yang pertama kali dibangkitkan, simpul tetangga yang dihasilkan harus ditulis ke dalam *cache*. *Cache* mencegah proses *scraping* berulang kali untuk suatu simpul.

Sebagai upaya semakin mempercepat program, terdapat *hashmap visited* yang digunakan dalam algoritma IDS. Struktur data ini digunakan untuk mencegah eksplorasi simpul yang sama dalam suatu batas kedalaman. Jika suatu simpul telah dieksplorasi, maka program tidak akan kembali mengeksplorasi simpul tersebut. Hal ini tentunya akan semakin menghemat komputasi yang dilakukan.

Upaya optimasi yang terakhir adalah dengan melakukan *task distribution*, dimana beberapa *process* dan *thread* akan melakukan eksplorasi graf secara bersamaan. Secara *default*, ada 100 *threads* yang akan memproses algoritma IDS secara bersamaan. Seratus IDS ini akan memproses *subtree* yang berbeda. Optimasi ini memungkinkan algoritma IDS untuk dipercepat secara signifikan.



Gambar 3.1.1.1 Ilustrasi pembagian *task* pada algoritma IDS

Dengan strategi algoritma IDS yang telah dipaparkan, didapat kompleksitas waktu algoritma ini adalah  $O(K * N)$  dan kompleksitas ruang  $O(N)$ , dengan K banyak iterasi batas kedalaman maksimum yang dibutuhkan hingga solusi ditemukan dan N banyak simpul yang dikunjungi.

### 3.1.2. Algoritma Breadth First Search

Pada algoritma BFS kami, digunakan 2 string *start* dan *end* sebagai parameter input yaitu link yang merupakan artikel awal serta link tujuan akhir. Terdapat 2 buah jenis algoritma BFS yang kami gunakan tergantung pada jenis solusi yang ingin ditemukan. Apabila pengguna hanya ingin mencari 1 buah

solusi tercepat, kami menggunakan list of string sebagai parameter output. Sedangkan untuk menemukan semua *path* solusi dengan kedalaman terendah akan dikeluarkan sebuah 2D list of string.

Untuk mendukung keberjalanan algoritma ini, kami menggunakan struktur data buatan untuk node pada graph permasalahan ini. Setiap node akan menyimpan level kedalaman pencarian dimana node itu ditemukan, serta list of string berupa seluruh parent dari node tersebut. Selain itu, terdapat pula 2 buah struktur data primitif yaitu *queue* dan *map*. Queue merupakan struktur data yang memungkinkan penyimpanan data secara FIFO (*First In, First Out*). Struktur ini digunakan untuk penyimpanan link yang akan dikunjungi selanjutnya pada sisi paling depan queue (*Top*). Sedangkan map, yang merupakan koleksi data berbentuk pasangan *key-value*, dimana key yang digunakan adalah URL dan value merupakan struktur data node yang diidentifikasi url tersebut. Dengan map ini pula kami dapat mengetahui apakah suatu node telah dilewati atau belum, hal ini sangat berpengaruh terhadap keefektifan algoritma BFS.

Pencarian diawali dengan memulai pencarian pada node *start*, dimana node ini memiliki level 0 dan tidak memiliki parent, lalu node akan dimasukkan kepada queue. Selanjutnya selama belum ditemukan solusi, queue akan melihat url yang berada paling depan untuk diproses terlebih dahulu, dengan menggunakan sebuah fungsi. Hal ini dilakukan sampai seluruh node yang berada pada level ini sudah di proses, baru pencarian dilanjutkan pada level kedalaman selanjutnya. Pada saat fungsi proses node, dijalankan ia akan membangkitkan seluruh simpul tetangga menggunakan *web scraper*. Masing-masing tetangga akan diperiksa apakah ia sudah dikunjungi sebelumnya. Apabila belum, maka ia akan dimasukkan pada ujung belakang queue. Program akan terus berjalan sampai node *end* diproses, dimana pencarian akan dihentikan.

Terdapat sedikit perbedaan pendekatan untuk pencarian salah satu solusi dengan pencarian seluruh solusi. Apabila kita hanya akan mencari sebuah solusi tercepat, fungsi proses node hanya perlu mencatat siapakah parent dari node tersebut. Akibatnya, untuk membangkitkan solusi, algoritma dapat berjalan mundur dari *end* menelusuri siapakah parentnya secara berulang sampai mencapai *start*. Untuk menemukan seluruh solusi, fungsi proses node tidak hanya mencatat siapakah parent yang membangkitkannya, ia juga perlu mencatat bahwa level kedalaman saat ia ditemukan bernilai 1 lebih dari level parentnya. Selanjutnya, perlu diamati pula apabila ia kembali diproses setelah dibangkitkan suatu node lain, ia perlu memeriksa apakah node yang memanggilnya memiliki level yang bernilai 1 kurang dari dirinya. Karena apabila hal itu benar, maka ia dapat dijadikan parent alternatif dari dirinya, sedangkan jika tidak parent tersebut tidak perlu dicatat karena akan mengakibatkan suatu

path yang tidak minimum (tidak langsung). Akibatnya, untuk membangkitkan semua *path* solusi dibutuhkan suatu algoritma baru menyerupai *Depth First Search*. Algoritma ini akan memulai pencarian dari node *end*, lalu menelusuri setiap parentnya satu persatu. Selama penelusuran, ia akan mencatat node apa saja yang ia kunjungi hingga ia mencapai node *start*, dimana akan terbentuk suatu path solusi. Dengan melakukan pencarian pada seluruh kemungkinan parent, akan diperoleh list semua kemungkinan path solusi untuk pencarian ini.

Sama seperti pada algoritma IDS diatas, kami melakukan optimasi dengan *task distribution*, dimana beberapa *process* dan *thread* akan melakukan eksplorasi graf secara bersamaan. Saat memanggil semua Top node dari queue pada suatu level tertentu, program akan menciptakan thread tersendiri untuk memproses masing-masing node dengan syntax bahasa Go yaitu go routine. Secara default akan terdapat 50 threads yang dijalankan. Untuk menghindari terjadinya *race condition*, algoritma BFS menggunakan wait group dan mutual exclusion dalam mengakses struktur data queue dan map. Akibatnya, risiko akibat multithreading dapat diminimalisir.

Dengan Algoritma BFS seperti yang telah dijelaskan diatas, algoritma ini memiliki kompleksitas waktu  $O(N)$ . Sedangkan kompleksitas ruang dari algoritma ini adalah  $O(N)$ . Dimana nilai  $N$  pada kasus ini adalah banyaknya simpul yang dikunjungi selama pencarian.

### 3.2. Langkah Pemetaan Masalah

Dalam persoalan *WikiRace* ini, digunakan *mapping* berikut dari elemen-elemen algoritma yang telah dibuat:

1. Artikel Wikipedia: artikel Wikipedia dipetakan menjadi sebuah simpul graf.
2. *Interlink* (*link* ke artikel lain) dalam sebuah artikel: dipetakan menjadi sebuah sisi dalam graf. Dengan kata lain, jika sebuah artikel A memiliki *link* ke artikel B, dikatakan simpul A memiliki sisi (*edge*) dengan simpul B pada graf. Perlu diperhatikan bahwa sisi ini bersifat satu arah (*directed edge*), karena mungkin ada suatu artikel A yang memuat *link* artikel B, namun artikel B tidak memuat *link* artikel A.

Untuk menemukan simpul tetangga dari suatu simpul (dengan kata lain artikel dengan *link* dalam suatu artikel), dilakukan proses *scraping*, dimana semua *link* dalam *webpage* artikel Wikipedia tersebut akan dikumpulkan dan digunakan dalam jalannya algoritma.

Dengan pemetaan tersebut, berikut adalah rancangan singkat dari algoritma yang kami buat:

- a. IDS: Dalam algoritma IDS, akan dicoba batas kedalaman dari 1 hingga 10, dan kemudian program akan memanggil prosedur DLS dengan batas

kedalaman yang telah ditentukan. Prosedur DLS akan melakukan penelusuran secara rekursif; jika simpul yang ingin dieksplorasi merupakan simpul tujuan, maka program akan mengembalikan *boolean* true yang akan diterima oleh prosedur utama dan digunakan untuk menentukan *reachability* simpul tujuan dengan batas kedalaman tersebut. Program utama kemudian akan mengembalikan batas kedalaman dan *path* yang ditemukan.

- b. BFS: Pada algoritma ini, akan dilakukan pencarian mulai dari node awal, dimana ia dimasukkan ke dalam *queue*. Selama simpul akhir belum ditemukan, *queue* akan mengeluarkan *link* yang terletak paling depan untuk diproses. Pada saat proses suatu simpul, ia akan membangkitkan seluruh simpul tetangganya dan mencatat dirinya sebagai parent mereka, lalu simpul tetangga dimasukkan pada belakang node. Proses akan berhenti apabila simpul akhir ditemukan, lalu setiap parent akan ditelusuri mundur dari node akhir untuk membangkitkan *path* solusi.

### 3.3. Arsitektur Aplikasi Web

Dalam mendesain aplikasi berbasis web sebagai antarmuka tugas besar ini, kelompok kami menimbang ada 4 poin desain yang harus dipenuhi:

- a. Aplikasi web memungkinkan pengguna untuk memasukkan dua judul artikel wikipedia sebagai simpul awal dan akhir
  - i. Aplikasi web dapat menyajikan rekomendasi artikel dari input yang dimaksud untuk mempermudah input
- b. Aplikasi web memungkinkan pengguna untuk memilih metode pencarian (IDS atau BFS) dan juga jumlah solusi yang diberikan (*single* atau *multiple*)
- c. Aplikasi web menunjukkan hasil jalur yang ditemukan, waktu yang dibutuhkan, serta jumlah artikel yang ditelusuri
  - i. Jalur yang ditemukan ditampilkan secara interaktif dalam bentuk *directed graph*
- d. Desain antarmuka aplikasi web nyaman dilihat

### 3.4. Contoh Ilustrasi Kasus

Untuk memperjelas implementasi algoritma, akan dipaparkan langkah-langkah penelusuran dengan algoritma IDS dan BFS dengan simpul awal: Yorgos Lanthimos dan simpul akhir: Lobster.

- a. Algoritma IDS:

Simpul awal [Salsa fuliginata](#) akan ditelusuri ketetanggaannya menggunakan proses *scraping*. Hasil dari proses *scraping* adalah sebuah *list* yang berisi *url* dari artikel simpul tetangga. Masing-masing *url* tersebut akan ditelusuri dengan cara rekursi. Proses secara lengkapnya dapat diperhatikan dalam tabel berikut:

**TABEL IDS**

Simpul Ekspan	Simpul Hidup (hanya ditampilkan 5 simpul hidup pertama)	Penjelasan
<a href="#"><u>Salsa fuliginata</u></a>	<a href="#"><u>Orb-Weaver spider</u></a> , <a href="#"><u>Salsa</u></a> , <a href="#"><u>Hatching</u></a> , <a href="#"><u>Carapace</u></a> , <a href="#"><u>Abdomen</u></a>	Prosedur dipanggil oleh prosedur utama dengan limit = 1
<a href="#"><u>Orb-Weaver spider</u></a>	<a href="#"><u>Salsa</u></a> , <a href="#"><u>Hatching</u></a> , <a href="#"><u>Carapace</u></a> , <a href="#"><u>Abdomen</u></a>	<b>BACKTRACK</b> Prosedur dipanggil dengan limit = 0. Karena limit = 0 dan simpul ekspan bukan tujuan, program <i>backtrack</i> .
<a href="#"><u>Salsa</u></a>	<a href="#"><u>Hatching</u></a> , <a href="#"><u>Carapace</u></a> , <a href="#"><u>Abdomen</u></a>	<b>BACKTRACK</b> Prosedur dipanggil dengan limit = 0. Karena limit = 0 dan simpul ekspan bukan tujuan, program <i>backtrack</i> .
<a href="#"><u>Hatching</u></a>	<a href="#"><u>Carapace</u></a> , <a href="#"><u>Abdomen</u></a>	<b>BACKTRACK</b> Prosedur dipanggil dengan limit = 0. Karena limit = 0 dan simpul ekspan bukan tujuan, program <i>backtrack</i> .
<a href="#"><u>Carapace</u></a>	<a href="#"><u>Abdomen</u></a>	<b>BACKTRACK</b> Prosedur dipanggil dengan limit = 0. Karena limit = 0 dan simpul ekspan bukan tujuan, program <i>backtrack</i> .

<a href="#"><u>Abdomen</u></a>	-	Prosedur dipanggil dengan limit = 0. Namun, karena simpul ekspan sama dengan tujuan, program kembali.
--------------------------------	---	-------------------------------------------------------------------------------------------------------

Tabel 3.4.1. Tabel IDS

b. Algoritma BFS:

Serupa seperti algoritma IDS, Simpul awal [Salsa fuliginata](#) akan ditelusuri ketetanggaannya menggunakan proses *scraping*. Hasil dari proses *scraping* adalah sebuah *list* yang berisi *url* dari artikel simpul tetangga. Masing-masing *url* tersebut akan ditelusuri dikunjungi satu-persatu sebagai level pertama pencarian. Proses selengkapnya adalah sebagai berikut.

**TABEL BFS**

Simpul Ekspan	Simpul Hidup (hanya ditampilkan 5 simpul hidup pertama)	Penjelasan
<a href="#"><u>Salsa fuliginata</u></a>	<a href="#"><u>Orb-Weaver spider</u></a> , <a href="#"><u>Salsa</u></a> , <a href="#"><u>Hatchling</u></a> , <a href="#"><u>Carapace</u></a> , <a href="#"><u>Abdomen</u></a>	Prosedur dipanggil oleh prosedur utama dengan level = 0
<a href="#"><u>Orb-Weaver spider</u></a>	<a href="#"><u>Salsa</u></a> , <a href="#"><u>Hatchling</u></a> , <a href="#"><u>Carapace</u></a> , <a href="#"><u>Abdomen</u></a>	Pencarian Level 1 pada simpul ekspan Orb-Weaver spider untuk menemukan simpul tetangganya pada level 2
<a href="#"><u>Salsa</u></a>	<a href="#"><u>Hatchling</u></a> , <a href="#"><u>Carapace</u></a> , <a href="#"><u>Abdomen</u></a>	Pencarian Level 1 pada simpul ekspan Salsa untuk menemukan simpul tetangganya pada level 2
<a href="#"><u>Hatchling</u></a>	<a href="#"><u>Carapace</u></a> , <a href="#"><u>Abdomen</u></a>	Pencarian Level 1 pada simpul ekspan Hatchling untuk menemukan simpul

		tetangganya pada level 2
<u>Carapace</u>	<u>Abdomen</u>	Pencarian Level 1 pada simpul ekspan Carapace untuk menemukan simpul tetangganya pada level 2
<u>Abdomen</u>	-	Telah ditemukan simpul Abdomen yang merupakan simpul tujuan, sehingga <i>path</i> solusi telah ditemukan

**Tabel 3.4.2.** Tabel BFS

## BAB 4

### Implementasi dan Pengujian

#### 4.1. Struktur Data, Fungsi dan Prosedur Algoritma IDS

Struktur data yang digunakan dalam algoritma IDS ini adalah sebagai berikut

Nama Variabel	Tipe	Penjelasan Kegunaan
IdsIterationState	<pre>type IdsIterationState struct {     MutexSet sync.RWMutex     Set      map[string]struct{} }</pre>	Digunakan untuk meng-keep <i>track</i> simpul yang telah dikunjungi. Mutex berguna supaya operasi <i>write</i> ke variabel Set berjalan tanpa <i>race condition</i> .
IdsGlobalState	<pre>type IdsGlobalState struct {     MutexLevel     sync.Mutex     Level         map[string]int     MutexCache   sync.RWMutex     Cache          map[string][]string     MutexParents sync.Mutex     Parents        map[string][]string }</pre>	Digunakan untuk meng-keep <i>track</i> level dari sebuah simpul yang telah ditemukan. Digunakan untuk menyimpan <i>cache</i> dari operasi <i>scraping</i> . Variable parents digunakan untuk menyimpan simpul tetangga dengan level kedalaman lebih rendah, yang kemudian akan digunakan untuk me-reconstruct jalur. Variabel mutex berfungsi untuk menjaga operasi <i>read</i> dan <i>write</i> dari <i>race condition</i> .
found	found chan bool	Channel yang digunakan oleh suatu proses untuk mengkomunikasikan bahwa suatu jalur telah ditemukan.
stopNow	stopNow chan bool	(Hanya digunakan di <i>single solution</i> ) Digunakan untuk memerintahkan seluruh <i>thread</i> yang sedang berjalan untuk berhenti.

availableProcess	<code>availableProcess chan bool</code>	Digunakan untuk berkomunikasi dengan fungsi utama ketika sebuah <i>process</i> selesai melakukan eksekusi
------------------	-----------------------------------------	-----------------------------------------------------------------------------------------------------------

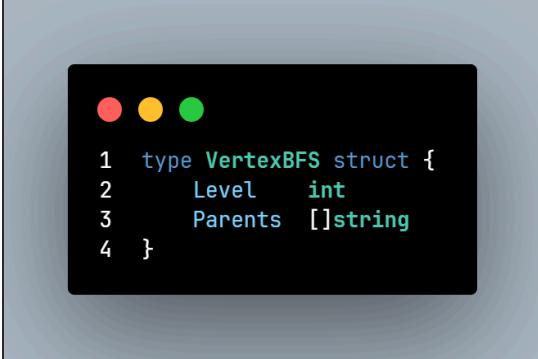
Fungsi dan prosedur yang diimplementasikan adalah sebagai berikut

Nama Fungsi/Prosedur	Tipe	Penjelasan Kegunaan
dfsSingleSolution	<code>func dfsSingleSolution(from, start, end string, limit, currentDepth, currentIter int, found chan&lt;- bool, counter *int, visited *models.IdsIterationState, globalLevel *models.IdsGlobalState, stopNow chan bool)</code>	Fungsi dfs yang digunakan dalam proses eksplorasi simpul <i>single solution</i> . Bersifat rekursif untuk mengeksplorasi simpul tetangga.
dfsMultipleSolution	<code>func dfsMultipleSolution(from, start, end string, limit, currentDepth, currentIter int, found chan&lt;- bool, counter *int, visited *models.IdsIterationState, globalLevel *models.IdsGlobalState)</code>	Fungsi dfs yang digunakan dalam proses eksplorasi simpul <i>multiple solution</i> . Bersifat rekursif untuk mengeksplorasi simpul tetangga.
idsSingleProcessSingleSolution	<code>func idsSingleProcessSingleSolution(realStart, start, end string, limit, currentDepth, currentIter int, found chan&lt;- bool, availableProcess chan&lt;- bool, counter *int, visited *models.IdsIterationState, globalLevel *models.IdsGlobalState, stopNow chan bool)</code>	Fungsi pembantu yang akan dipanggil oleh setiap <i>threads</i> yang akan melakukan eksplorasi simpul. Memanggil prosedur dfsSingleSolution
idsSingleProcessMultipleSolution	<code>func idsSingleProcessMultipleSolution(realStart, start, end string, limit, currentDepth, currentIter int, found chan&lt;- bool, availableProcess chan&lt;- bool, counter *int, visited</code>	Fungsi pembantu yang akan dipanggil oleh setiap <i>threads</i> yang akan melakukan eksplorasi simpul. Memanggil prosedur dfsMultipleSolution

	<pre>*models.IdsIterationState, globalLevel *models.IdsGlobalState)</pre>	
IdsSingleSolution	<pre>func IdsSingleSolution(start, end string) models.Response</pre>	Fungsi utama yang akan dipanggil oleh <i>frontend</i> untuk metode IDS dengan <i>single solution</i> .
IdsMultipleSolution	<pre>func IdsMultipleSolution(start, end string) models.Response</pre>	Fungsi utama yang akan dipanggil oleh <i>frontend</i> untuk metode IDS dengan <i>multiple solution</i> .

## 4.2. Struktur Data, Fungsi dan Prosedur Algoritma BFS

Untuk algoritma BFS, kami menggunakan struktur data sebagai berikut.

Nama Variabel	Tipe	Penjelasan Kegunaan
VertexBFS	 <pre>1 type VertexBFS struct { 2     Level    int 3     Parents  []string 4 }</pre>	Digunakan sebagai struktur data yang merepresentasikan suatu node. Atribut level merupakan tingkat kedalaman dalam pencarian dimana ia ditemukan. Di lain pihak, parents merupakan slice of string yang digunakan untuk mencatat seluruh parent node tersebut.
QueueSafe	 <pre>1 type QueueSafe struct { 2     Queue  Queue[string] 3     Mutex  sync.Mutex 4     Map    map[string]VertexBFS 5 }</pre>	Struktur data yang digunakan dalam algoritma multiple solution. Digunakan agar terhindar dari race condition saat melakukan multithreading. Mutex atau mutual exclusion merupakan variabel dari package "sync" yang akan memastikan tidak terjadinya race condition. Queue dan Map merupakan struktur data utama yang

		digunakan pada algoritma ini. Map pada struktur ini memetakan sebuah string URL terhadap VertexBFS yang direpresentasikan nya
QueueSingle	<pre> 1 type QueueSingle struct { 2     Queue []string 3     Map   map[string]string 4     Mutex sync.Mutex 5 } </pre>	Serupa seperti struktur data QueueSafe, namun digunakan pada pencarian single solution. Perbedaan utamanya adalah pada tipe Map. Untuk struktur ini, Map memetakan string URL pada suatu string lainnya yang melambangkan URL parent.

Fungsi dan Prosedur yang diimplementasikan untuk algoritma ini adalah sebagai berikut

Nama Fungsi/Prosedur	Tipe	Penjelasan Kegunaan
BFSSingleSolution	<pre> 1 func BFSSingleSolution(start, end string) models.Response </pre>	Fungsi utama yang menjalankan proses algoritma BFS untuk mencari hanya 1 solusi. Parameter input fungsi ini adalah kedua string start dan end, yang merupakan URL mulai dan selesai permainan wikirace. Output dari fungsi ini berbentuk model Response yang akan digunakan api.go dan dikirimkan pada front end.

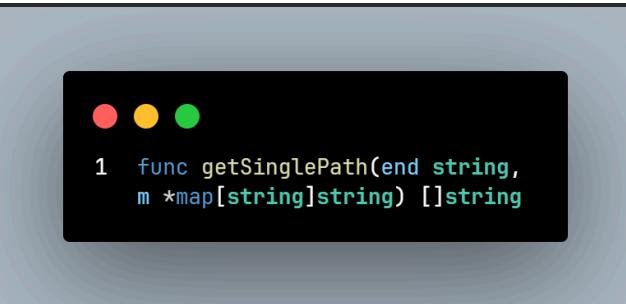
processSingleVerte  
x



```
● ● ●
1 func processSingleVertex(curr string, WG *sync.WaitGroup,  
QS *models.QueueSingle)
```

Prosedur yang dipanggil oleh BFSSingleSolution ketika ingin memproses sebuah node. Prosedur ini akan mencari seluruh simpul tetangga dari simpul yang di proses, lalu dimasukkan pada queue seperti yang dijelaskan sebelumnya. Parameter input terdiri atas curr yaitu string URL yang diproses, WG yaitu pointer kepada WaitGroup dari package “sync”, dan QS sebuah pointer kepada QueueSingle

getSinglePath



```
● ● ●
1 func getSinglePath(end string,  
m *map[string]string) []string
```

Fungsi yang digunakan untuk mengeluarkan sebuah *path* solusi. Fungsi ini akan dipanggil pada akhir BFSSingleSolution. Parameter input terdiri atas end yaitu string URL tujuan, m pointer menuju map dari QueueSingle. Fungsi mengembalikan sebuah slice string yaitu *path* solusi.

BFSMultipleSolution	 <pre>1 func BFSMultipleSolution(start, end string) models.Response</pre>	<p>Fungsi utama yang menjalankan proses algoritma BFS untuk mencari semua solusi. Parameter input fungsi ini adalah kedua string start dan end, yang merupakan URL mulai dan selesai permainan wikirace. Output dari fungsi ini berbentuk model Response yang akan digunakan api.go dan dikirimkan pada front end.</p>
processCurrentVertex	 <pre>1 func processCurrentVertex(curr string, QS *models.QueueSafe, WG *sync.WaitGroup)</pre>	<p>Prosedur yang dipanggil oleh BFSMultipleSolution ketika ingin memproses sebuah node. Prosedur ini akan mencari seluruh simpul tetangga dari simpul yang di proses, lalu dimasukkan pada queue seperti yang dijelaskan sebelumnya. Parameter input terdiri atas curr yaitu string URL yang diproses, WG yaitu pointer kepada WaitGroup dari package “sync”, dan QS sebuah pointer kepada QueueSafe</p>

getAllPath

```
1 func getAllPaths(curr string, M *map[string]
models.VertexBFS, tempMap *map[string][][]string)
```

Fungsi yang digunakan untuk mengeluarkan semua *path* solusi yang mungkin. Fungsi ini akan dipanggil pada akhir BFSMultipleSolution . Parameter input terdiri atas end yaitu string URL tujuan, M pointer menuju map dari QueueSingle, serta tempMap pointer ke sebuah map yang menyimpan semua path yang dimulai dari suatu node. Fungsi mengembalikan sebuah 2D slice of string yang merupakan kumpulan semua *path* solusi.

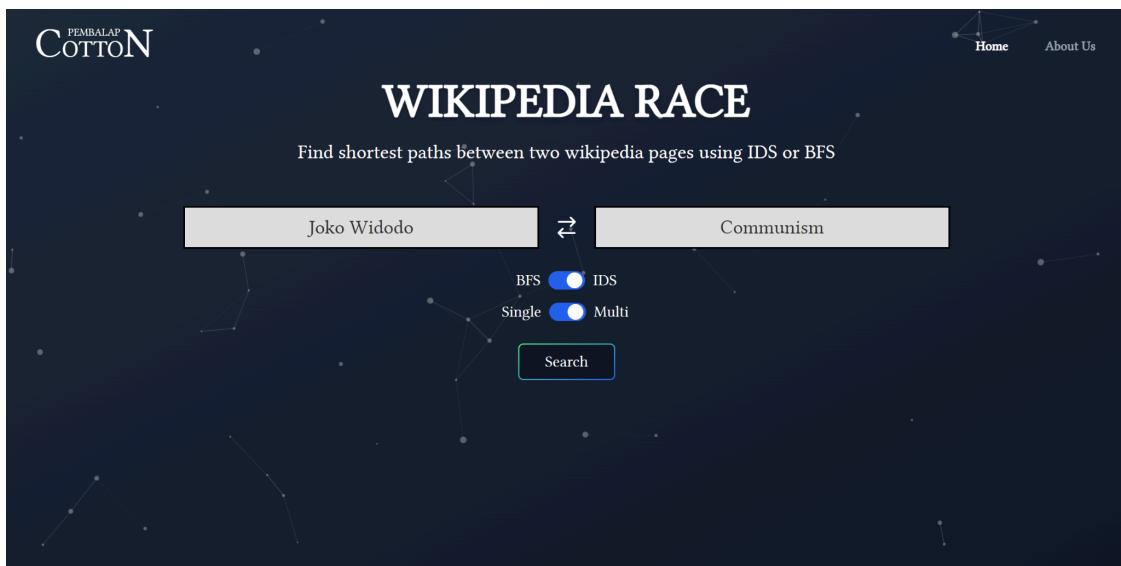
#### 4.3. Tata Cara Penggunaan Program

Berikut adalah tata cara penggunaan program.

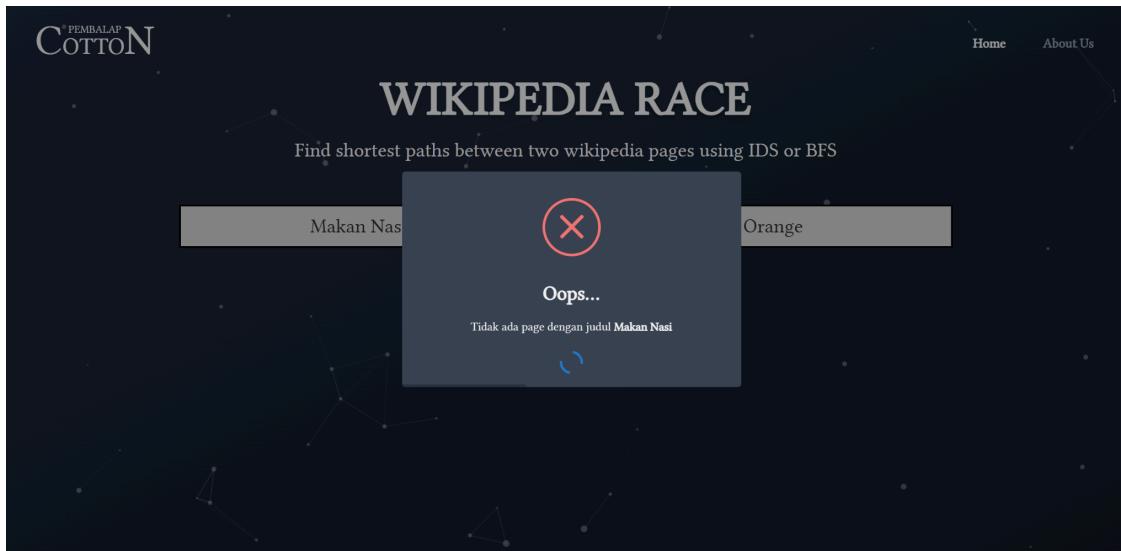
1. Masukkan judul artikel awal pada *textbox* bagian kiri, dan judul artikel tujuan pada *textbox* bagian kanan. Pada saat pengisian judul, program akan memberikan rekomendasi judul artikel wikipedia yang paling cocok dengan masukan pengguna sehingga pengguna dapat melihat apakah terdapat judul dengan masukan pengguna.



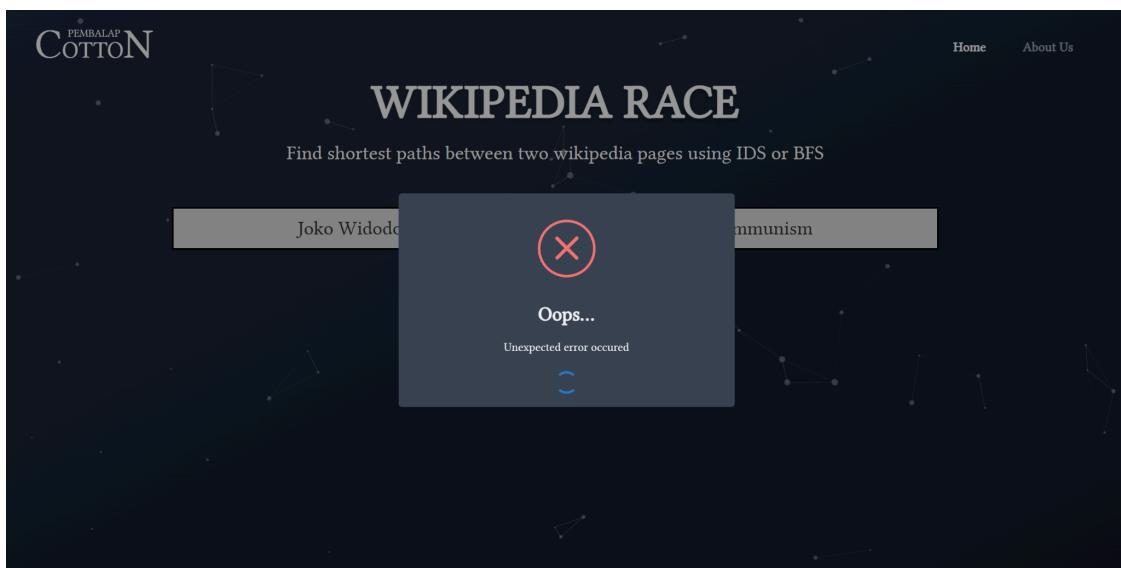
2. Pilih jenis algoritma dan banyak solusi yang diinginkan.



3. Klik tombol search untuk melihat hasil pencarian jarak terpendek antara artikel awal dan tujuan.
4. Jika judul yang dimasukkan pengguna tidak ada pada wikipedia, program akan memberikan pesan error bahwa judul tidak tersedia.



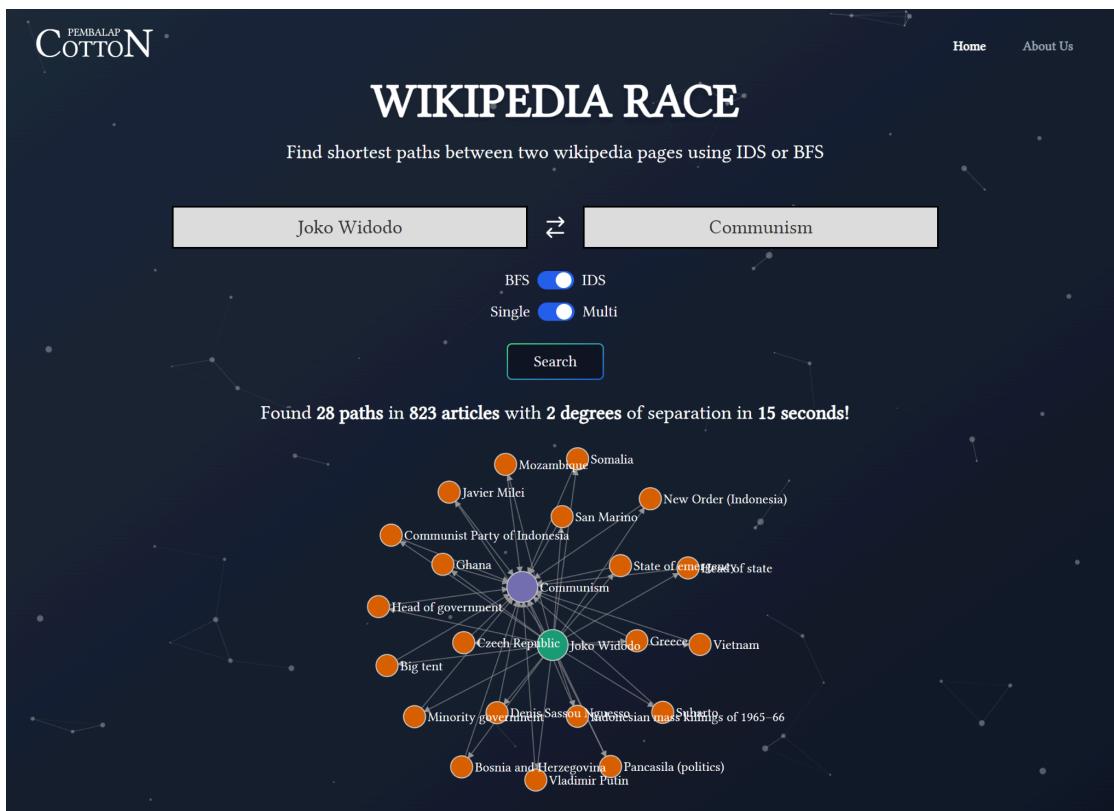
5. Jika terdapat error seperti error jaringan, program akan memberikan pesan sebagai berikut.



6. Jika tidak terdapat kesalahan pada judul artikel dan error lainnya, program akan melakukan kalkulasi dan akan terlihat proses *loading* pada tombol *search*.



7. Jika kalkulasi sudah selesai, program akan menampilkan graf hasil yang memiliki fitur *zoom-in* dan *zoom-out*. Jika salah satu *node* diklik, artikel yang direferensikan oleh *node* tersebut akan terbuka pada *browser* pengguna.



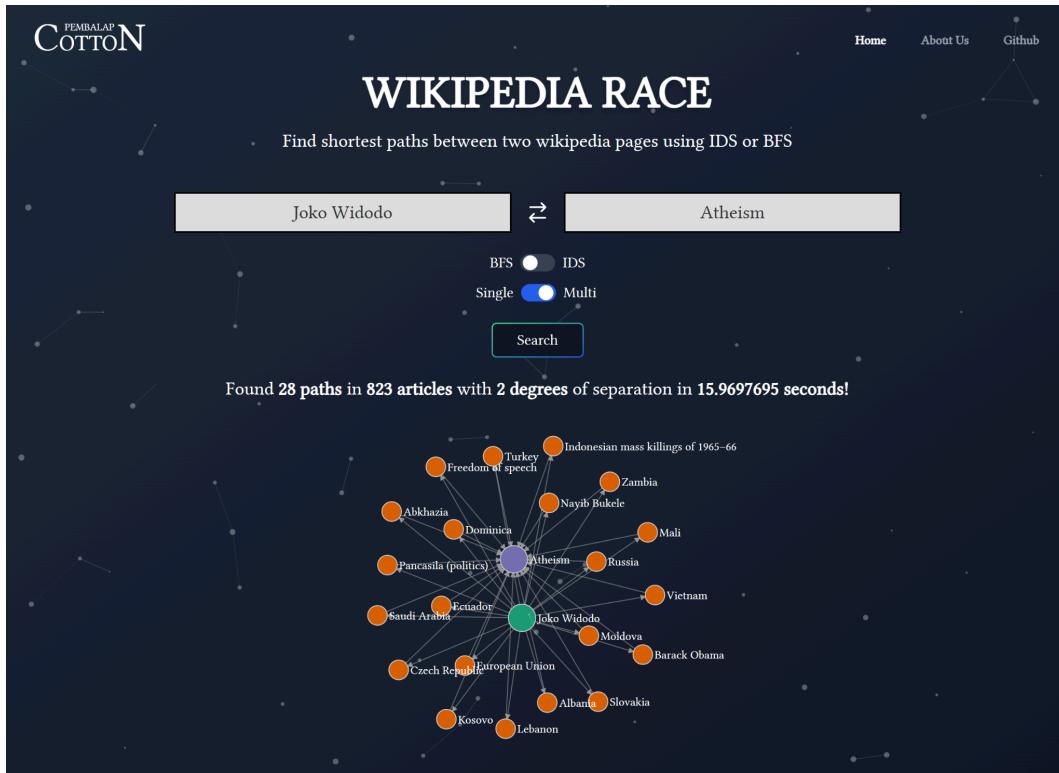
## 4.4. Hasil Pengujian

### A. Artikel Joko Widodo -> Atheism

#### A.1. BFS Single Path



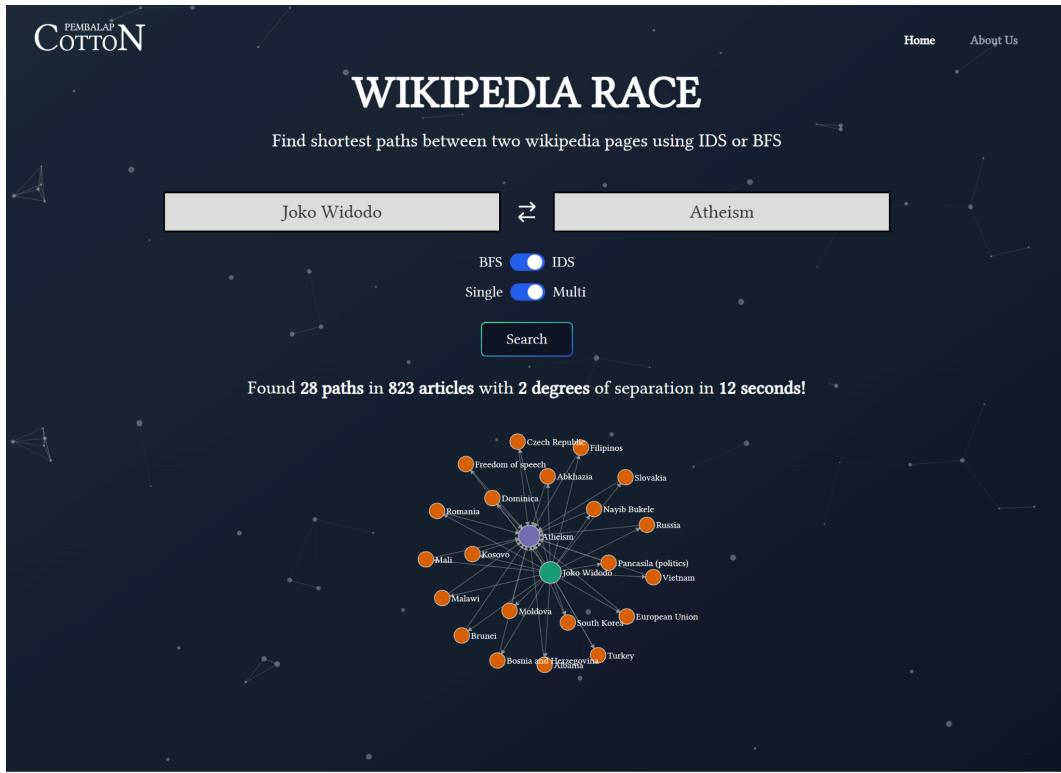
## A.2 BFS Multiple Paths



## A.3 IDS Single Path



#### A.4 IDS Multiple Paths

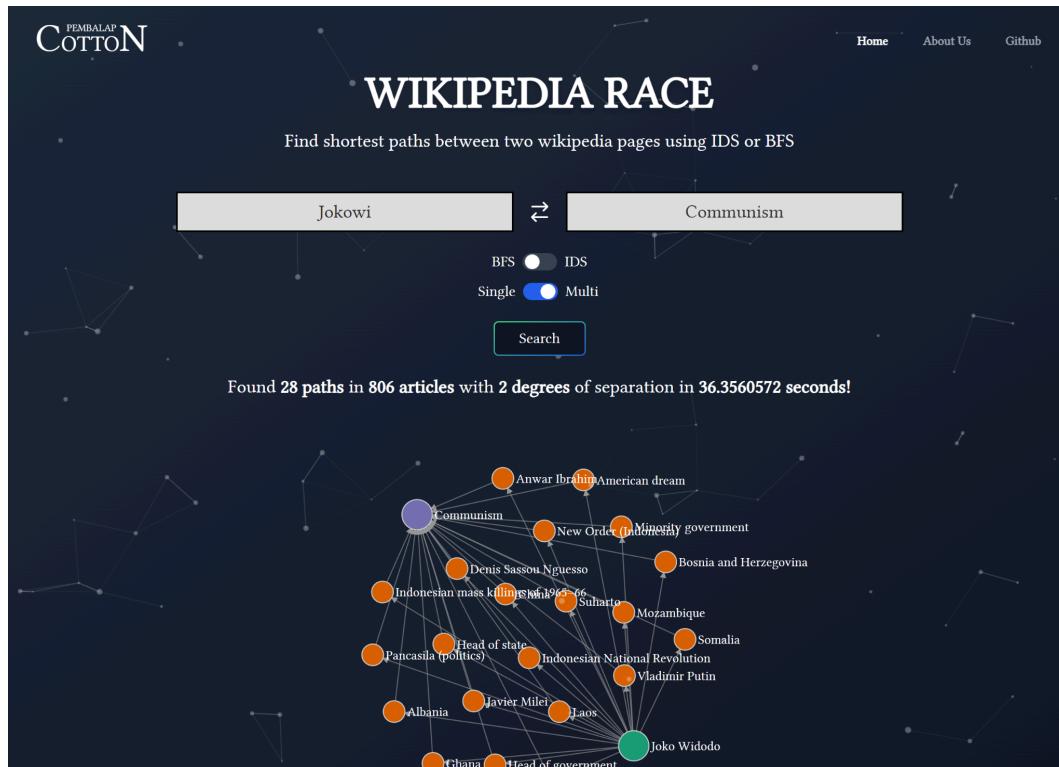


#### B. Artikel Joko Widodo -> Communism

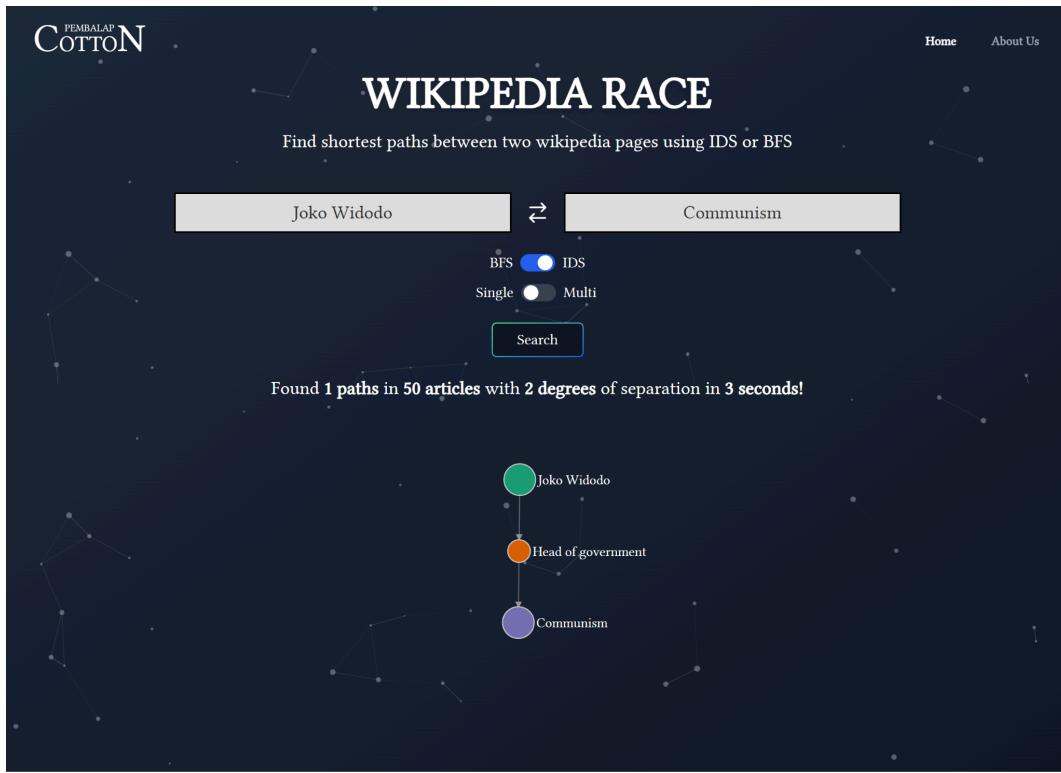
##### B.1 BFS Single Path



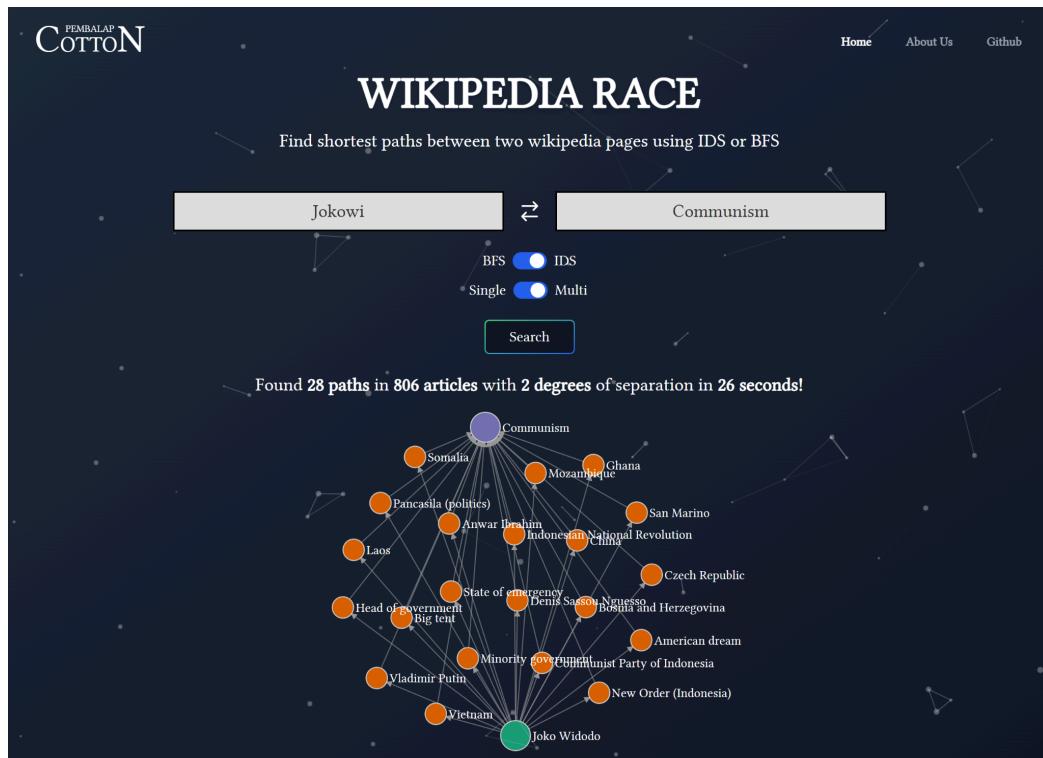
## B.2 BFS Multiple Paths



## B.3 IDS Single Path

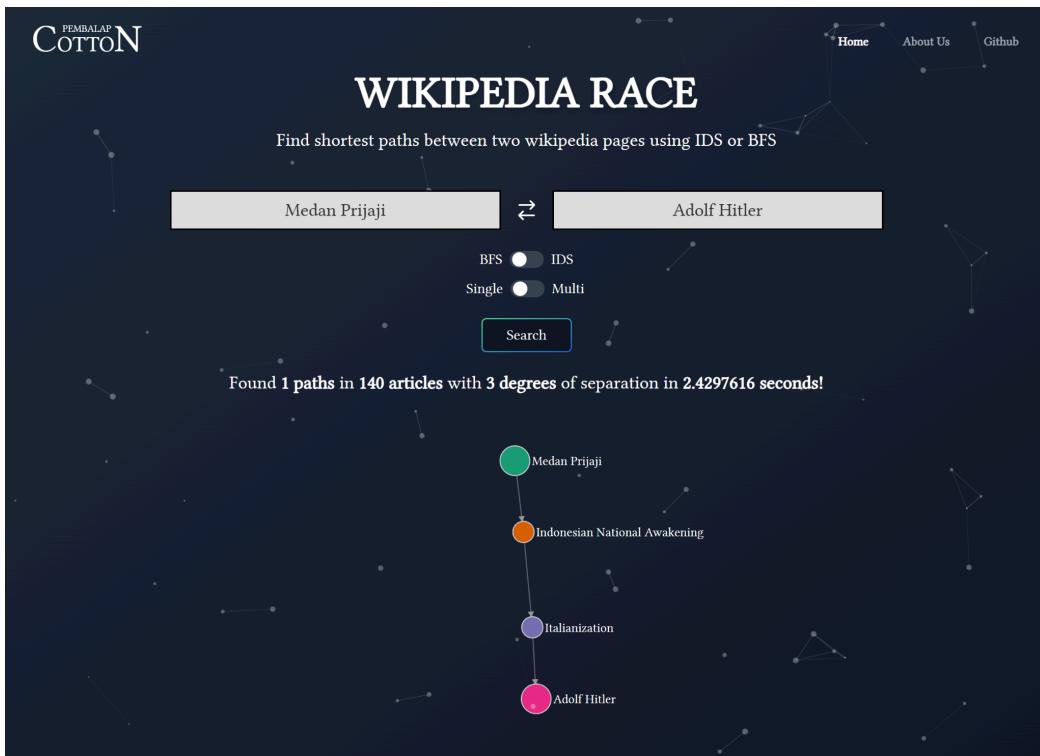


#### B.4 IDS Multiple Path

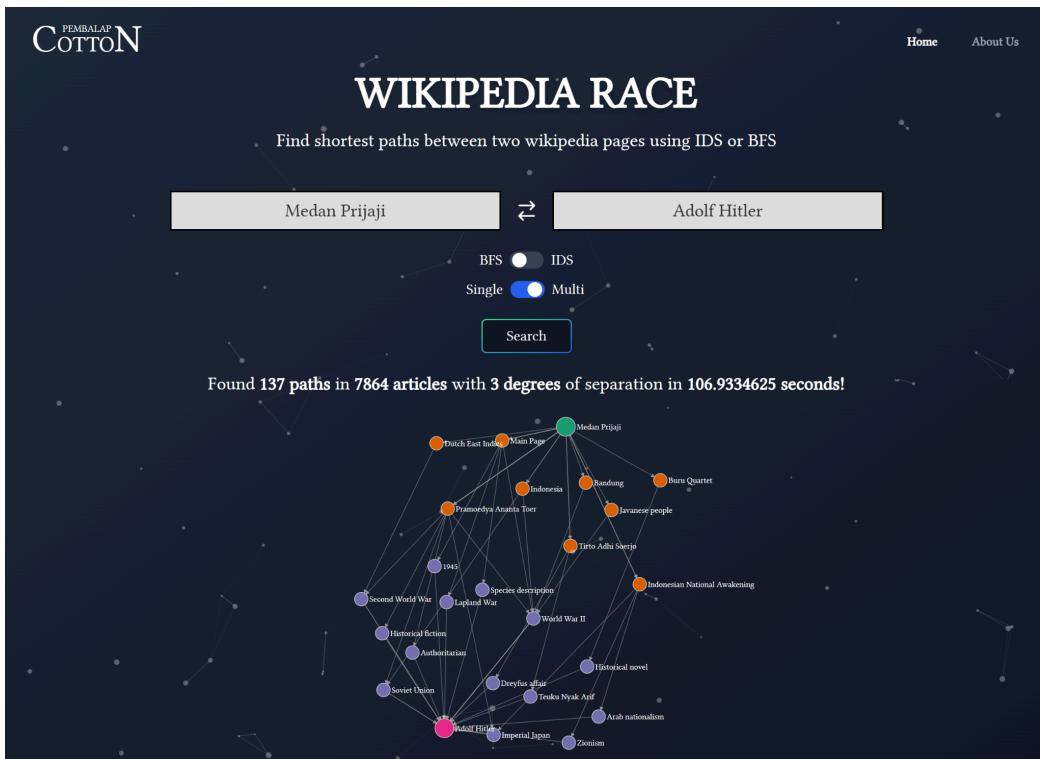


#### C. Medan Prijaji -> Adolf Hitler

### C.1 BFS Single Path



### C.2 BFS Multiple Paths



### C.3 IDS Single Path



#### C.4 IDS Multiple Paths



#### 4.5. Analisis Hasil Pengujian

No	Derajat	Algoritma	Banyak Solusi	Waktu (s)
1	2	BFS	Tunggal	0.5594
2	2	BFS	Banyak	15.97
3	2	IDS	Tunggal	2
4	2	IDS	Banyak	26
5	2	BFS	Tunggal	1.08
6	2	BFS	Banyak	36
7	2	IDS	Tunggal	3
8	2	IDS	Banyak	26
9	3	BFS	Tunggal	2.42
10	3	BFS	Banyak	106
11	3	IDS	Tunggal	1
12	3	IDS	Banyak	100

Berdasarkan pengujian tersebut, dapat dilihat bahwa program dapat menemukan jarak terpendek antara artikel awal ke artikel akhir dalam waktu rata-rata kurang dari 1 menit, terutama untuk *single solution*. Di sisi lain, pencarian banyak solusi memakan waktu lebih banyak karena harus melakukan pencarian terhadap semua *node* pada *level/degree* yang sama dengan solusi yang pertama ditemukan. Selain itu, pencarian *single path* pada IDS memiliki kemungkinan untuk selesai lebih cepat. Hal ini karena IDS menggunakan konsep DFS yang secara umum akan menemukan simpul tetangga yang dekat ke *root*. Dapat dilihat bahwa untuk seluruh eksplorasi dengan solusi tunggal, program berjalan kurang dari 1 menit.

Jika diperhatikan, banyak jalur yang ditemukan berubah-ubah dari waktu ke waktu. Hal ini dikarenakan proses *scraping* yang gagal, sehingga tidak mengembalikan seluruh *link* pada *webpage*. Untuk memastikan proses jalannya program yang benar, diperlukan *bandwidth* internet yang besar.

Dapat diperhatikan juga bahwa jumlah artikel yang dikunjungi jumlahnya konsisten, hal ini merupakan reasuransi bahwa kedua algoritma berjalan dengan baik.

## BAB 5

# Penutup

### 5.1. Kesimpulan

Pada Tugas Besar 2 Strategi Algoritma ini, kelompok kami telah membuat *solver* dari permainan *WikiRace* dengan menggunakan algoritma *Iterative Deepening Search* dan *Breadth-First Search*. *Solver* yang dibuat memiliki *interface* berbasis web, dengan bagian *frontend* dibangun TypeScript dan JavaScript, sedangkan bagian *backend* dibuat dengan bahasa pemrograman Go. Program kami dapat menemukan jalur tunggal (*single path*) dalam waktu kurang dari 1 menit dan jalur banyak (*multiple path*). Pengguna akan disajikan *path* yang ditemukan oleh program, waktu yang diperlukan, serta banyak artikel yang dikunjungi (dilakukan *scraping*). Pengguna juga dapat melihat dan berinteraksi dengan graf jalur yang ditemukan.

### 5.2. Saran

Sebaiknya spesifikasi Tugas Besar 2 Strategi Algoritma diperlengkap, terutama pada restriksi-restriksi yang diperbolehkan dan dilarang. Ada baiknya juga *degree* maksimum dari *test case* yang akan diujikan dituliskan dalam spesifikasi, mengingat komputasi yang dilakukan untuk setiap *degree* kurang lebih bertambah secara eksponensial. Kelompok kami juga ingin memberikan apresiasi atas bimbingan asisten yang diberikan asisten Strategi Algoritma dalam menjawab pertanyaan di *sheets QnA*.

### 5.3. Refleksi

Algoritma IDS dan BFS pada algoritma *blind search*, dimana tidak ada informasi tambahan yang dapat digunakan dalam proses eksplorasi simpul. Untuk melakukan optimasi selanjutnya, dapat dilakukan *informed search*, dimana program menggunakan informasi tambahan untuk mempercepat jalannya algoritma. Algoritma yang menerapkan metode *informed search* antara lain A-star, dan Greedy Best First Search.

Kelompok kami dari Tugas Besar 2 Strategi Algoritma ini juga merasakan bahwa I/O (lebih tepatnya proses *scraping*), membutuhkan waktu yang sangat lama. Hal ini dibuktikan dengan observasi bahwa *bottleneck* program ada pada proses *scraping*.

## Lampiran

### 1. Checklist

Poin	Ya	Tidak
Program berhasil dijalankan	✓	
Program dapat menampilkan jarak terpendek tunggal dari artikel awal ke artikel tujuan menggunakan algoritma IDS dan BFS	✓	
[Bonus] Program dapat menampilkan seluruh jarak terpendek (tidak hanya satu) dan memvisualisasikannya	✓	
[Bonus] Program dapat mencari rute terpendek dengan durasi kurang dari satu menit	✓	
[Bonus] Program dijalankan menggunakan Docker baik untuk frontend maupun backend	✓	
[Bonus] Membuat video kelompok	✓	

### 2. Pranala Repository Front-End

[https://github.com/ninoaddict/Tubes2\\_FE\\_Pembalap-Kapas](https://github.com/ninoaddict/Tubes2_FE_Pembalap-Kapas)

### 3. Pranala Repository Back-End

[https://github.com/Farhannr28/Tubes2\\_BE\\_Pembalap-Kapas](https://github.com/Farhannr28/Tubes2_BE_Pembalap-Kapas)

### 4. Pranala Video

<https://youtu.be/etoN08leBYI>

### 5. Pembagian Tugas

Nama	Nim	Pembagian Tugas
Kristo Anugrah	13522024	idsSingleSolution.go, idsMultipleSolution.go, scraper.go, pengerojaan laporan, pengerojaan video
Farhan Nafis Rayhan	13522037	bfs, setup backend (API), pengerojaan laporan

Adril Putra Merin	13522068	frontend, setup docker, pengerjaan laporan, pengerjaan video
-------------------	----------	--------------------------------------------------------------------

## Daftar Pustaka

- Bu, Z., & Korf, R. E. (2022). *Iterative-deepening uniform-cost heuristic search*. Proceedings of the International Symposium on Combinatorial Search, 15(1), 20–28. <https://doi.org/10.1609/socs.v15i1.21748>
- Cook, D. J., & Varnell R. C. (1998). *Adaptive Parallel Iterative Deepening Search*. Journal of Artificial Intelligence Research, 9(1998), 139-166. <https://arxiv.org/pdf/1105.5447.pdf>
- Meta Open Source. (2024). Quick Start - React. <https://react.dev/learn>
- Munir, R., & Maulidevi, N. U. (2021). *Breadth/depth First Search (BFS/DFS)*. Breadth/Depth First Search (BFS/DFS). <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf>
- Observable. (n.d.). D3 Documentation. <https://d3js.org/getting-started>
- Tailwind Labs Inc. (2024). Tailwind CSS Documentation. <https://tailwindcss.com/docs/installation>
- Tauber, A., & Smith, V. (2020). Colly. <https://github.com/gocolly/colly>
- Vercel. (2024). Next.js Documentation. <https://nextjs.org/docs>