IF2211 - Strategi Algoritma

Laporan Tugas Besar 2

Pemanfaatan Algoritma BFS dan DFS dalam Pencarian Recipe pada Permainan Little Alchemy 2

Disusun Oleh:

Alfian Hanif Fitria Yustanto	13523073
Heleni Gratia M. Tampubolon	13523107
Ahmad Wicaksono	13523121

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2025

DAFTAR ISI

BAB 1
DESKRIPSI TUGAS

1.1 Deskripsi Tugas

Gambar 1. Little Alchemy 2
(sumber: https://www.thegamer.com)

Little Alchemy 2 merupakan permainan berbasis web / aplikasi yang dikembangkan oleh Recloak yang dirilis pada tahun 2017, permainan ini bertujuan untuk membuat 720 elemen dari 4 elemen dasar yang tersedia yaitu air, earth, fire, dan water. Permainan ini merupakan sekuel dari permainan sebelumnya yakni Little Alchemy 1 yang dirilis tahun 2010.

Mekanisme dari permainan ini adalah pemain dapat menggabungkan kedua elemen dengan melakukan drag and drop, jika kombinasi kedua elemen valid, akan memunculkan elemen baru, jika kombinasi tidak valid maka tidak akan terjadi apa-apa. Permainan ini tersedia di web browser, Android atau iOS.

Pada Tugas Besar kedua mata kuliah Strategi Algoritma ini, mahasiswa diminta untuk mengembangkan sebuah aplikasi pencarian recipe elemen dalam permainan Little Alchemy 2 dengan menerapkan strategi pencarian Breadth First Search (BFS) dan Depth First Search (DFS).

Komponen-komponen dari permainan ini antara lain:

1. Elemen dasar

Dalam permainan Little Alchemy 2, terdapat 4 elemen dasar yang tersedia yaitu water, fire, earth, dan air, 4 elemen dasar tersebut nanti akan di-combine menjadi elemen turunan yang berjumlah 720 elemen.

Gambar 2. Elemen dasar pada Little Alchemy 2

2. Elemen turunan

   Terdapat 720 elemen turunan yang dibagi menjadi beberapa tier tergantung tingkat kesulitan dan banyak langkah yang harus dilakukan. Setiap elemen turunan memiliki recipe yang terdiri atas elemen lainnya atau elemen itu sendiri.

3. Combine Mechanism

   Untuk mendapatkan elemen turunan pemain dapat melakukan combine antara 2 elemen untuk menghasilkan elemen baru. Elemen turunan yang telah didapatkan dapat digunakan kembali oleh pemain untuk membentuk elemen lainnya.

1.2 Spesifikasi

Pada Tugas Besar kedua mata kuliah Strategi Algoritma ini, mahasiswa diminta untuk mengembangkan sebuah aplikasi pencarian recipe elemen dalam permainan Little Alchemy 2 dengan menerapkan strategi pencarian Breadth First Search (BFS) dan Depth First Search (DFS). Komponen-komponen utama dalam tugas ini meliputi:

1. **Elemen dasar dan recipe**

   Dalam permainan Little Alchemy 2, pemain dapat menggabungkan dua elemen

untuk menghasilkan elemen baru. Aplikasi yang dibangun harus dapat menemukan urutan kombinasi (recipe) untuk menghasilkan elemen tertentu, dimulai dari elemen dasar.

2. **Algoritma pencarian**

   Aplikasi harus menyediakan opsi pemilihan strategi pencarian yang digunakan, yaitu BFS atau DFS. Untuk bonus, mahasiswa diperbolehkan menambahkan algoritma pencarian bidirectional.

3. **Mode pencarian**

   Terdapat dua mode pencarian yang dapat dipilih pengguna:

   - **Pencarian recipe terpendek**, yaitu mencari urutan kombinasi terpendek untuk mencapai elemen tujuan.
   - **Pencarian multiple recipe**, yaitu mencari beberapa jalur berbeda untuk menghasilkan elemen yang sama. Pengguna dapat menentukan jumlah maksimum recipe yang ingin dicari. Mode ini wajib dioptimasi dengan menggunakan multithreading untuk meningkatkan performa pencarian.

4. **Visualisasi recipe sebagai tree**

   Aplikasi harus menampilkan hasil pencarian dalam bentuk visualisasi tree, yang memperlihatkan struktur kombinasi elemen dari elemen dasar hingga ke elemen tujuan. Visualisasi ini diharapkan dapat membantu pengguna memahami langkah-langkah pembentukan elemen secara intuitif.

5. **Platform pengembangan**

   Aplikasi bersifat web-based dan dibangun dengan teknologi sebagai berikut:

   - **Frontend** menggunakan JavaScript dengan framework **Next.js** atau **React.js**.
   - **Backend** menggunakan bahasa pemrograman **Golang**.

     Struktur repository frontend dan backend boleh digabung atau dipisah sesuai kebutuhan tim.

6. **Sumber data**

   Data elemen dan resep yang dibutuhkan dalam permainan dapat diperoleh melalui teknik web scraping dari website Fandom Little Alchemy 2.

7. **Ketentuan pengerjaan**

   Tugas dikerjakan secara berkelompok dengan jumlah anggota minimal 2 orang dan maksimal 3 orang. Anggota kelompok diperbolehkan berasal dari kelas atau kampus yang berbeda.

BAB 2
LANDASAN TEORI

2.1 Traversal Graf

Traversal graf adalah proses mengunjungi simpul-simpul (nodes) dalam suatu graf secara sistematis. Traversal ini menjadi dasar dari banyak algoritma pencarian dan pemecahan masalah yang dapat direpresentasikan dalam bentuk graf. Graf sendiri adalah struktur data yang merepresentasikan hubungan antar objek, yang terdiri atas himpunan simpul (vertices) dan sisi (edges) yang menghubungkannya.

Dalam konteks pemecahan masalah berbasis graf, traversal dapat dianggap sebagai eksplorasi ruang solusi, di mana simpul-simpul mewakili kemungkinan keadaan (state) dan sisi-sisi mewakili transisi antar keadaan tersebut. Tujuan dari traversal ini adalah untuk menemukan jalur dari simpul awal (initial state) ke simpul tujuan (goal state).

Algoritma traversal graf dibagi menjadi dua kategori utama, yaitu pencarian melebar (*breadth first search*/BFS) dan pencarian mendalam (*depth first search*/DFS). Berdasarkan informasi yang digunakan selama pencarian, algoritma ini juga dapat diklasifikasikan menjadi dua jenis

1. Tanpa informasi (*uninformed/blind search*)

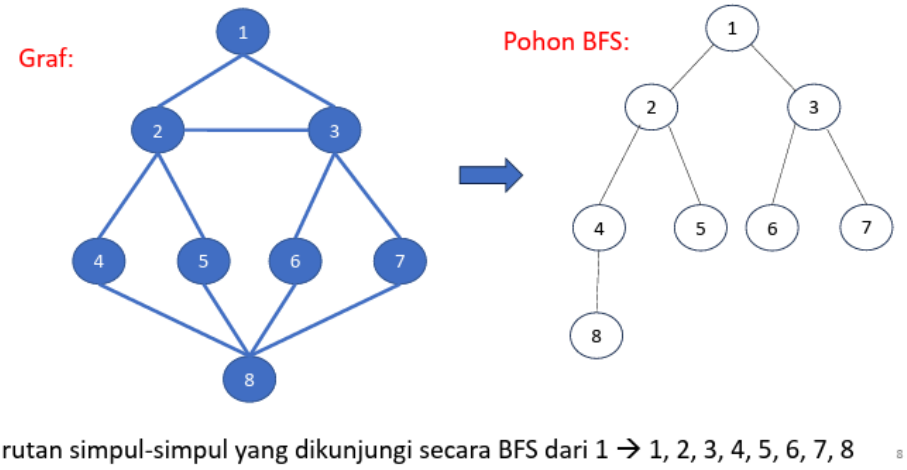
   Algoritma ini akan melakukan pencarian tanpa ada informasi tambahan yang disediakan. Contohnya DFS,BFS, Depth Limited Search, Iterative Deepening Search, dan Uniform Cost Search.
2. Dengan informasi (*informed search*)

   Pencarian yang dilakukan berbasis heuristik dan mengetahui *non-goal state* yang lebih menjanjikan daripada yang lain sehingga dapat mempercepat proses pencarian. Contohnya Best First Search dan A*.

Selain itu, dalam proses traversal, graf dapat direpresentasikan secara statis maupun dinamis. Graf statis yaitu graf yang sudah terbentuk sebelum proses pencarian dilakukan. Graf tersebut direpresentasikan sebagai struktur data. Sebaliknya, graf dinamis yaitu graf yang terbentuk saat proses pencarian dilakukan.

2.2. Breadth First Search (BFS)

Breadth-First Search (BFS) merupakan salah satu algoritma traversal graf yang paling dasar dan umum digunakan. Algoritma ini digunakan untuk menelusuri atau mencari simpul dalam graf secara sistematis dengan pendekatan melebar, yaitu mengunjungi semua simpul yang bertetangga terlebih dahulu sebelum melanjutkan ke simpul-simpul di tingkat berikutnya. BFS sangat berguna dalam berbagai aplikasi, terutama untuk mencari jalur terpendek dalam graf tak berbobot.



Gambar 3. Pembentukan Pohon BFS
(sumber:https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf)

Dalam penerapannya, BFS memiliki prinsip kerja sebagai berikut:

1. **Inisialisasi:**
   - Pilih simpul awal (source).
   - Masukkan simpul tersebut ke dalam antrian (queue).
   - Tandai simpul tersebut sebagai telah dikunjungi menggunakan array Boolean.
2. **Eksplorasi:**

   Selama antrian tidak kosong, maka dilakukan:
   - Ambil (dequeue) simpul dari antrian.
   - Kunjungi simpul tersebut (misalnya dengan mencetak nilainya atau menyimpannya ke dalam hasil).
   - Untuk setiap tetangganya yang belum dikunjungi:
     1. Masukkan ke dalam antrian.
     2. Tandai sebagai telah dikunjungi.

3. **Terminasi:**
    - Proses berhenti ketika antrian kosong, yang berarti semua simpul yang dapat dijangkau dari simpul awal telah dikunjungi.

Struktur data yang digunakan dalam algoritma ini terdiri dari:

1. **Antrian (Queue):** Untuk menyimpan daftar simpul yang akan dikunjungi berikutnya.
2. **Array Boolean** dikunjungi: Untuk menandai simpul mana saja yang telah dikunjungi agar tidak dikunjungi berulang kali.
3. **Matriks Ketetanggaan** ($A = [a_{i□}]$):

    a. Matriks berukuran $n \times n$.

    b. $a_{i□} = 1$ jika simpul *i* dan simpul *j* bertetangga, 0 jika tidak.

BFS memiliki karakteristik penting yaitu traversal dilakukan berdasarkan level. Artinya, simpul yang berada paling dekat (dalam jumlah langkah minimum) dari simpul awal akan dikunjungi lebih dahulu. Hal ini menjadikan BFS sangat cocok untuk permasalahan pencarian jalur terpendek dalam graf tak berbobot.

Keunggulan BFS antara lain:

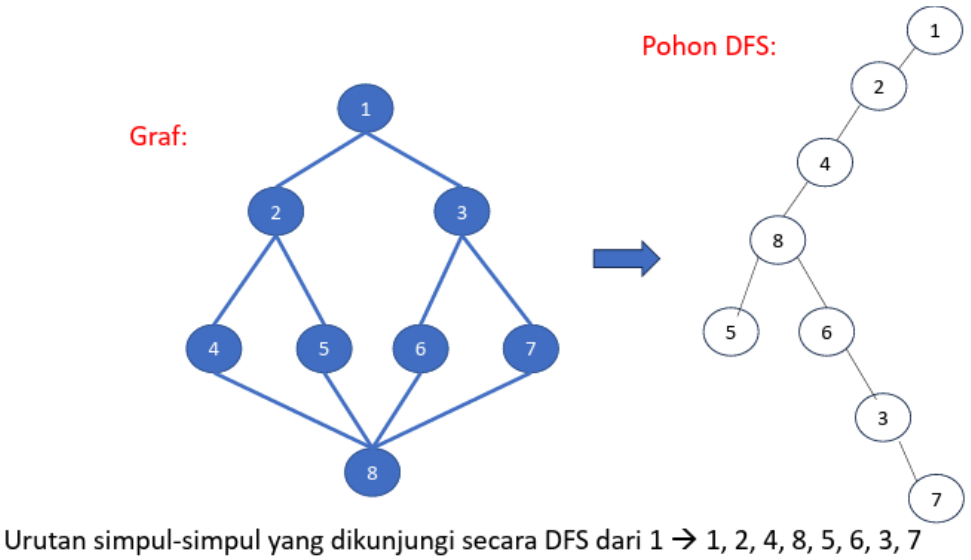
1. Dapat menemukan solusi/jalur terpendek secara optimal pada graf tak berbobot.
2. Cocok untuk pencarian dalam ruang-ruang masalah yang memiliki banyak simpul di setiap levelnya.

Namun, BFS juga memiliki kekurangan, yaitu:

1. Membutuhkan memori yang besar karena semua simpul pada level yang sama harus disimpan dalam antrian sebelum berpindah ke level berikutnya.
2. Kurang efisien jika solusi berada jauh dari simpul awal dan struktur graf sangat dalam

2.3. Depth First Search (DFS)

Depth-First Search (DFS) merupakan algoritma traversal graf yang bekerja dengan prinsip mendalam, yaitu menelusuri satu jalur hingga akhir sebelum berpindah ke jalur lain. Dalam algoritma ini, setiap simpul akan dikunjungi sedalam mungkin terlebih dahulu, baru kemudian dilakukan penelusuran ke simpul lain yang sejajar. Konsep DFS ini mirip dengan traversal *preorder* pada pohon, di mana kita menyelesaikan penelusuran satu cabang sepenuhnya sebelum beralih ke cabang lainnya.



Gambar 4. Pembentukan Pohon DFS
(sumber:https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf)

Secara umum, proses DFS bekerja sebagai berikut:

1. **Inisialisasi:**
   - Pilih simpul awal (*start vertex*), misalnya v.
   - Tandai simpul tersebut sebagai telah dikunjungi menggunakan array Boolean.
   - Panggil DFS secara rekursif untuk setiap tetangga yang belum dikunjungi.
2. **Eksplorasi:**
   - Kunjungi simpul v.
   - Pilih salah satu simpul w yang bertetangga dengan v dan belum dikunjungi.
   - Lanjutkan penelusuran DFS dari simpul w.
   - Jika mencapai simpul u yang semua tetangganya telah dikunjungi, maka lakukan *backtrack* ke simpul sebelumnya yang masih memiliki tetangga yang belum dikunjungi.
3. **Terminasi:**
   - Proses akan berakhir jika tidak ada lagi simpul yang belum dikunjungi dan dapat dicapai dari simpul awal.

Dalam algoritma Depth-First Search (DFS), struktur data yang digunakan untuk mendukung proses traversal adalah tumpukan (stack), baik secara eksplisit maupun implisit. Pada implementasi yang bersifat rekursif, penggunaan stack tidak tampak secara langsung karena fungsi rekursi secara otomatis memanfaatkan stack pemanggilan fungsi (*call stack*) dari sistem untuk menyimpan urutan simpul yang sedang dijelajahi. Sementara itu, pada implementasi iteratif, struktur stack digunakan secara eksplisit oleh programmer untuk melacak simpul-simpul yang akan dikunjungi berikutnya. Dengan demikian, baik secara langsung maupun tidak, DFS selalu mengandalkan prinsip kerja stack—*Last In, First Out (LIFO)*—untuk melakukan penelusuran secara mendalam dalam graf.

Selain itu, algoritma ini juga memanfaatkan:

1. **Array Boolean dikunjungi:** Untuk menandai simpul yang telah dikunjungi dan mencegah siklus tak hingga pada graf yang mengandung *cycle*.
2. **Matriks Ketetanggaan atau List Ketetanggaan:** Untuk menyimpan hubungan antar simpul.

DFS memiliki beberapa karakteristik penting, yaitu:

1. Cocok untuk graf dalam (deep graph): DFS sangat efisien dalam menyusuri struktur graf yang dalam dan memiliki sedikit cabang.
2. Penggunaan memori lebih hemat: Karena DFS tidak perlu menyimpan semua simpul pada tingkat tertentu secara bersamaan seperti BFS.
3. Dapat digunakan untuk berbagai keperluan: seperti deteksi siklus, topological sort, dan mencari komponen terkoneksi dalam graf.

Namun, DFS juga memiliki beberapa keterbatasan, berupa:

1. Tidak menjamin solusi/jalur terpendek ditemukan pada graf tak berbobot.
2. Jika graf sangat luas dan dalam, penggunaan rekursi dapat menyebabkan *stack overflow*.

2.4. Pengembangan *Website*

Pengembangan website adalah proses yang mencakup berbagai aktivitas teknis untuk membangun, merancang, dan memelihara sebuah situs web. Aktivitas ini melibatkan pembuatan halaman web menggunakan bahasa markup seperti HTML, serta integrasi dengan teknologi pemrograman lainnya seperti CSS, JavaScript, dan bahasa pemrograman *server-side*.

Website pada dasarnya merupakan kumpulan halaman digital yang disimpan di komputer khusus yang terhubung dengan internet selama 24 jam, yang disebut dengan web server. Berdasarkan cara pengelolaan dan penyajian kontennya, situs web dapat dikategorikan menjadi dua jenis utama:

1. **Static Website**

   Merupakan situs web dengan halaman yang sudah dibangun sebelumnya dan disajikan secara langsung oleh server tanpa adanya proses pengolahan dinamis berdasarkan permintaan pengguna. Konten disusun menggunakan HTML, CSS, atau JavaScript dan tidak terhubung dengan basis data. Karena tidak memerlukan pemrosesan server, situs statis cenderung lebih cepat dan lebih murah dalam pengelolaan.

2. **Dynamic Website**

   Halaman situs ini dibentuk secara dinamis saat diminta oleh pengguna, biasanya menggunakan bahasa pemrograman sisi server seperti PHP dan Node.js. Situs ini mampu berinteraksi dengan basis data, memungkinkan pembaruan konten secara mudah dan efisien. Walaupun performanya lebih lambat dibandingkan situs statis, fleksibilitas dan skalabilitasnya jauh lebih tinggi.

Secara umum, pengembangan web terbagi menjadi dua bagian utama, yaitu pengembangan front-end dan back-end.

2.4.1. Pengembangan Web *Front-End*

Pengembangan web *front-end* adalah proses yang berfokus pada transformasi data dan informasi menjadi antarmuka pengguna yang dapat diakses dan digunakan dengan mudah. Proses ini mengandalkan penggunaan teknologi seperti HTML (HyperText Markup Language), CSS (Cascading Style Sheets), dan JavaScript untuk menyusun elemen-elemen visual yang interaktif, termasuk teks, warna, gambar, ikon, tombol, hingga navigasi menu.

Seorang *front-end developer* memiliki tanggung jawab untuk memastikan tampilan dan pengalaman pengguna (*User Experience* atau UX) berjalan dengan optimal. Mereka mewujudkan rancangan visual menjadi kode program, membangun serta memelihara *User Interface* (UI), dan mengoptimalkan tampilan untuk berbagai perangkat termasuk desktop maupun seluler dengan pendekatan desain responsif.

2.4.2. Pengembangan Web *Back-End*

Berbeda dari pengembangan front-end yang berurusan langsung dengan tampilan visual, pengembangan web back-end berfokus pada logika, sistem pemrosesan data, serta arsitektur server yang mendukung fungsi internal dari sebuah situs web. Bagian ini berjalan di sisi server dan tidak dapat diakses langsung oleh pengguna, namun memiliki peran vital sebagai tulang punggung dari setiap aplikasi web. Tanggung jawab utama dari back-end developer meliputi penyimpanan dan pengelolaan data, pengaturan komunikasi dengan basis data, serta penghubung antara komponen antarmuka pengguna dengan sistem back-end melalui API (Application Programming Interface) atau layanan web lainnya.

Dalam implementasinya, pengembangan back-end sering kali melibatkan penggunaan bahasa pemrograman server-side seperti PHP, Node.js, Python, Ruby, atau Java. Terdapat dua pendekatan utama dalam perancangan sistem back-end, yakni *Object-Oriented Programming* (OOP) dan *Functional Programming*. OOP menekankan pada pembuatan objek dan struktur data yang berinteraksi satu sama lain dalam urutan logis tertentu. Sementara itu, pendekatan fungsional lebih fleksibel dalam urutan eksekusinya dan sering digunakan dalam bidang analisis data, seperti pada bahasa pemrograman SQL, R, dan F#.

Meskipun tidak tampak oleh pengguna, pengembangan back-end sangat menentukan kestabilan, keamanan, dan performa situs web secara keseluruhan. Tanpa sistem *back-end* yang andal, situs web tidak akan mampu memproses data pengguna, menyimpan informasi penting, maupun menjalankan logika bisnis dengan baik.

2.5. *Programming Language*

2.5.1. Go Language (Golang)

Bahasa Go, atau yang juga dikenal sebagai Golang, adalah bahasa pemrograman tingkat tinggi yang dikembangkan oleh Google pada tahun 2007 dan diumumkan sebagai proyek open-source pada tahun 2009. Bahasa ini dirancang untuk menjawab berbagai tantangan dalam pengembangan perangkat lunak modern, seperti waktu kompilasi yang lambat, manajemen dependensi yang tidak terkendali, dan kesulitan dalam membangun perangkat lunak lintas bahasa (*cross-language development*).

Go mengusung paradigma statically typed (penentuan tipe data dari variabel yang dilakukan saat waktu kompilasi) dan compiled language, namun tetap menjaga kesederhanaan seperti yang ditemukan pada bahasa pemrograman dinamis seperti Python. Pendekatan ini memungkinkan pengembang menulis program yang efisien, bersih, dan mudah dipelihara, tanpa mengorbankan kecepatan dan performa aplikasi. Kompilasi kode dalam Go berlangsung sangat cepat dan langsung menghasilkan kode mesin, tanpa memerlukan *virtual machine (VM)* seperti pada Java. Selain itu, Go telah dilengkapi dengan *garbage collector* dan *runtime reflection*, memberikan kenyamanan tambahan dalam pengelolaan memori dan struktur data secara dinamis.

Salah satu kekuatan utama dari Go adalah mekanisme konkuren (concurrency) yang dibangun secara native melalui *goroutines*—proses ringan yang memungkinkan program untuk menjalankan banyak tugas secara bersamaan atau paralel. Hal ini menjadikan Go sangat cocok digunakan dalam pengembangan aplikasi jaringan, layanan cloud, sistem terdistribusi, hingga arsitektur *microservices*, yang membutuhkan efisiensi tinggi dalam menangani banyak proses secara simultan.

Selain fitur sintaksis yang ekspresif dan ringkas, Go juga menawarkan perpustakaan standar (standard library) yang sangat lengkap, sehingga pengembang tidak perlu bergantung pada pustaka eksternal untuk keperluan umum seperti manipulasi file, jaringan, encoding, dan lain sebagainya. Manajemen dependensi pun didesain agar efisien melalui sistem *packages*, sehingga struktur proyek dapat lebih modular dan mudah dikendalikan.

Secara keseluruhan, Go merupakan bahasa pemrograman yang menggabungkan kekuatan dari bahasa-bahasa terdahulu seperti C dan Python, namun hadir dengan pendekatan baru yang mengedepankan produktivitas, kecepatan, dan skalabilitas. Karena alasan-alasan inilah, Go semakin banyak diadopsi dalam dunia industri dan akademik, terutama untuk pengembangan sistem modern yang membutuhkan kinerja tinggi dan pengelolaan paralelisme yang optimal.

2.5.2. ReactJS

React.js adalah pustaka JavaScript yang digunakan untuk membangun antarmuka pengguna berbasis komponen secara efisien dan deklaratif. Dikembangkan oleh Facebook, React memungkinkan pengembang membuat UI yang interaktif dengan cara

membagi antarmuka menjadi komponen-komponen kecil yang menerima data (props) dan mengembalikan elemen visual yang ditampilkan di layar.

Komponen React dapat merespons interaksi pengguna, seperti input teks, dengan memperbarui data dan merefleksikannya langsung ke tampilan. React juga mendukung integrasi bertahap, di mana pengembang bisa menambahkan komponen React ke dalam halaman HTML yang sudah ada.

Selain pustaka UI, React juga merupakan arsitektur yang mendukung pengambilan data secara asinkron dari file, database, atau server, lalu meneruskannya ke komponen interaktif. Pendekatan ini menjadikan React fleksibel dalam membangun aplikasi web modern, baik secara client-side maupun server-side.

Salah satu fitur kunci React adalah Virtual DOM (Document Object Model). Alih-alih memperbarui elemen-elemen pada DOM secara langsung setiap kali data berubah, React menggunakan representasi virtual dari DOM. Ketika terjadi perubahan data, React menghitung perbedaan (*diffing*) antara DOM virtual lama dan yang baru, lalu hanya menerapkan perubahan yang benar-benar diperlukan ke DOM asli. Proses ini dikenal sebagai *reconciliation* dan menghasilkan kinerja yang lebih cepat dan efisien, terutama pada aplikasi skala besar.

2.5.3. Docker

Docker adalah platform terbuka yang digunakan untuk mengembangkan, mengirimkan, dan menjalankan aplikasi di dalam lingkungan terisolasi yang disebut container. Dengan Docker, aplikasi dan semua yang dibutuhkannya dibungkus dalam satu container, jadi saat dijalankan di komputer lain (seperti server), tidak akan ada masalah karena semua sudah lengkap di dalamnya.

Container Docker bersifat ringan dan memuat semua yang dibutuhkan untuk menjalankan aplikasi, termasuk kode, runtime, pustaka, dan dependensi lainnya. Hal ini memastikan bahwa aplikasi berjalan sama, terlepas dari lingkungan tempat dijalankannya. Docker juga memungkinkan pengelolaan infrastruktur dilakukan dengan cara yang sama seperti mengelola aplikasi, serta mendukung kolaborasi tim melalui berbagi container yang seragam.

BAB 3
ANALISIS PEMECAHAN MASALAH

3.1. Langkah-Langkah Pemecahan Masalah

Permasalahan yang akan diselesaikan pada tugas besar kali ini adalah pencarian rute untuk membentuk suatu elemen target dari berbagai elemen yang tersedia. Pengguna dapat memasukkan masukan nama target elemen dan algoritma pemrosesan yang ingin dipilih, yaitu berupa Breadth-First Search (BFS) atau Depth-First Search (DFS).

Proses diawali dengan penerapan konsep *web scraping* untuk mengambil data seluruh elemen beserta elemen pembentuknya dan tingkat (*tier*) masing-masing. Hasil *scraping* akan disimpan dalam sebuah file .json dan dilakukan setiap kali website dimuat untuk pertama kalinya. Informasi tersebut kemudian disimpan ke dalam *slice* (struktur data serupa array yang lebih fleksibel dalam bahasa Go) untuk diproses lebih lanjut.

Pencarian rute dilakukan menggunakan algoritma yang dipilih. Pada BFS, pencarian dilakukan secara menyeluruh berdasarkan level, sedangkan pada DFS, pencarian dilakukan secara mendalam pada satu node terlebih dahulu sebelum menelusuri cabang lainnya. Pencarian akan mengevaluasi seluruh kemungkinan pohon (*tree*) yang terbentuk hingga ditemukan solusi yang sesuai dengan jumlah resep yang diinginkan. Setelah pencarian selesai, sistem akan menampilkan visualisasi pohon hasil pencarian, jumlah node yang dikunjungi, serta waktu yang dibutuhkan untuk menyelesaikan proses tersebut.

Program diimplementasikan menggunakan website dengan frontend menggunakan framework ReactJS serta backend menggunakan bahasa Go.

3.2. Mapping Persoalan sebagai Elemen BFS dan DFS

Dari persoalan yang ingin dipecahkan dengan menggunakan algoritma BFS dan DFS, terdapat elemen yang dapat dipetakan, yaitu nama elemen (root) sebagai simpul. Mapping elemen-elemen dari algoritma yang akan dibuat berdasarkan pemecahan masalah ini adalah:

a. Elemen dasar : simpul graf
b. Elemen target: simpul graf
c. ‘Root’: simpul graf
d. Left/Right : sisi graf

Dengan elemen-elemen tersebut, rancangan algoritma BFS dan DFS untuk menyelesaikan permasalahan ini, yaitu sebagai berikut:

a. Algoritma BFS

Algoritma ini menggunakan pendekatan *Breadth-First Search (BFS)* secara iteratif untuk menelusuri dan mencari semua kemungkinan resep dari sebuah elemen target berdasarkan struktur *recipeMap*. Proses pencarian dimulai dengan antrean (*queue*) yang menyimpan nama elemen root. Pada setiap iterasi, elemen yang berada di antrian paling depan akan diproses terlebih dahulu, memastikan eksplorasi dilakukan secara melebar sebelum menelusuri level lebih dalam. Untuk menghindari eksplorasi berulang dan menjamin efisiensi, diterapkan struktur visited berupa map yang menandai elemen yang sudah dikunjungi. Apabila ditemukan resep yang valid dan sesuai dengan batas *tierLimit*, maka elemen tersebut ditambahkan ke dalam daftar resep hasil. Pemrosesan dilanjutkan dengan memasukkan elemen-elemen pembentuknya (*left* dan *right*) ke dalam antrian.

Setelah semua resep ditemukan, proses rekonstruksi pohon dilakukan secara paralel menggunakan goroutine untuk tiap resep yang valid. Dalam setiap rekonstruksi, dilakukan backtracking untuk membangun semua kemungkinan pohon dari kombinasi elemen penyusun resep secara rekursif, dengan tetap menjaga status kunjungan node menggunakan salinan visited map yang terpisah untuk setiap pemanggilan. Untuk menjamin bahwa hasil akhir tidak melebihi batas jumlah pohon yang diminta oleh pengguna (*limit*), diterapkan pengaturan batas lokal pada setiap subtree, terutama pada pemrosesan bagian kiri (*left*) terlebih dahulu. Jika jumlah kombinasi subtree kiri sudah mencukupi, maka proses kombinasi dengan bagian kanan (*right*) akan dibatasi, menjaga agar total kombinasi tetap sesuai. Validitas subtree ditentukan berdasarkan keberadaan base component (*air, earth, fire, water*) atau apakah elemen tersebut memiliki resep turunannya. Jika tidak valid, maka subtree tersebut tidak diproses lebih lanjut.

b. Algoritma DFS

chro

3.3. Fitur Fungsional dan Arsitektur Aplikasi Web

3.3.1. Fitur Fungsional

Kode	Fungsionalitas	Deskripsi
F01	Masukan element	P/L memiliki interface untuk memasukkan string nama element pada website
F02	Masukkan jumlah resep yang ingin dicari	P/L memiliki interface untuk memasukkan integer jumlah resep
F03	Pencarian Query dengan DFS	P/L memiliki interface untuk melakukan query pencarian resep elemen menggunakan metode DFS
F04	Pencarian Query BFS	P/L memiliki interface untuk melakukan query pencarian resep elemen menggunakan metode BFS
F05	Scrap	P/L memiliki kemampuan untuk melakukan scraping
F06	Penampilan Recipe Tree	P/L memiliki kemampuan untuk memvisualisasikan hasil resep yang telah dibangun
F07	Paginasi Recipe Tree	P/L memiliki kemampuan untuk mengganti visualisasi resep jika terdapat lebih dari satu resep yang ditemukan
F08	Informasi Resep	P/L memiliki kemampuan untuk menampilkan informasi terkait hasil query resep

3.3.2. Arsitektur Aplikasi Web

Arsitektur aplikasi web dibedakan menjadi dua yaitu front-end dan back-end. Front-end berfungsi untuk menampilkan interface kepada pengguna. Back-end berfungsi untuk menangani proses algoritma dan mengirimkan hasilnya kepada front-end.

3.3.2.1. Front-End

Aplikasi front-end dibangun menggunakan bahasa java script dan memanfaatkan react.js sebagai *framework*, tailwind untuk tampilan , dan react-d3-tree untuk menampilkan graf. Berikut struktur program *front-end*.

```
.
├── Dockerfile
├── README.md
├── docker-compose.yml
├── package-lock.json
├── package.json
├── postcss.config.js
├── public
│   └── index.html
├── src
│   ├── App.js
│   ├── components
│   │   └── RecipeTree.js
│   ├── index.css
│   ├── index.js
│   └── pages
│       └── Home.js
└── tailwind.config.js
```

Pembangunan *front-end* memanfaatkan docker agar aplikasi dapat berbentuk container yang mudah digunakan. Tampilan utama program terletak pada file pages/Home.js dan code utama program terletak pada index.js . Component RecipeTree.js berisi komponen pembentuk graf pohon.

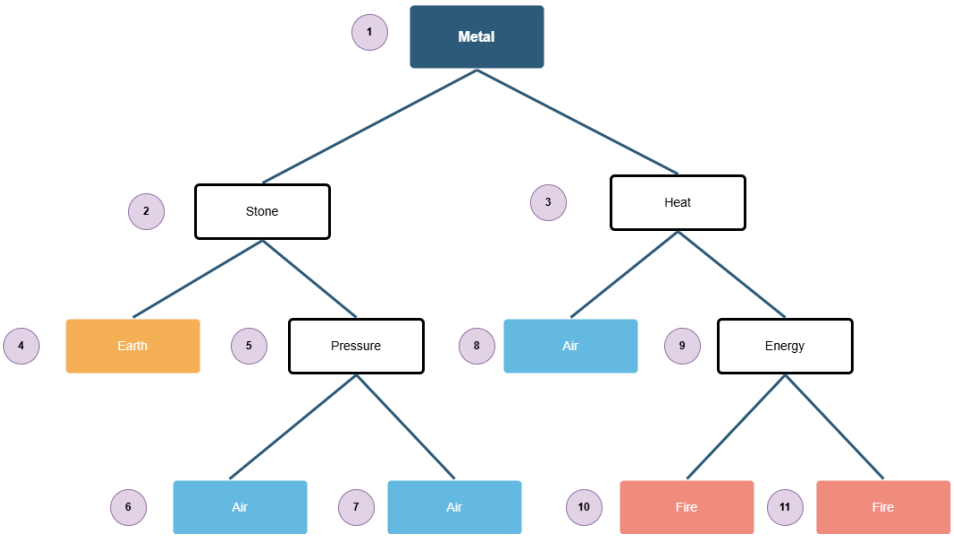
3.3.2.2. Back-End

Aplikasi *backend* dibangun menggunakan bahasa Golang. Aplikasi backend bertanggung jawab untuk melakukan scraping, dan pencarian resep. *Back-end* terdiri dari 5 struktur utama yaitu : main.go sebagai file utama yang meng-*handle* http *request*, folder handler yang berfungsi untuk melakukan handling terhadap request dalam bentuk JSON dan mengirimkan hasil *query* dalam format JSON, folder DFS yang berfungsi untuk menyimpan algoritma pencarian resep secara DFS, folder BFS yang berfungsi untuk menyimpan algoritma pencarian resep dengan metode bfs, folder scrapper yang berfungsi untuk mengatasi algoritma *scraper*.

```
.
├── BFS
│   └── MultipleRecipeBFS.go
├── DFS
│   └── MultipleRecipeDFS.go
├── Dockerfile
├── Element
│   ├── Element.go
│   └── Tree.go
├── Handler
│   ├── BFSHandler.go
│   ├── DFSHandler.go
│   └── ScrapperHandler.go
├── README.md
├── docker-compose.yml
├── go.mod
├── go.sum
├── main.go
└── scrapper
    └── scrapper.go
```

3.4. Contoh Ilustrasi Kasus

3.4.1 Breadth-First Search (BFS)



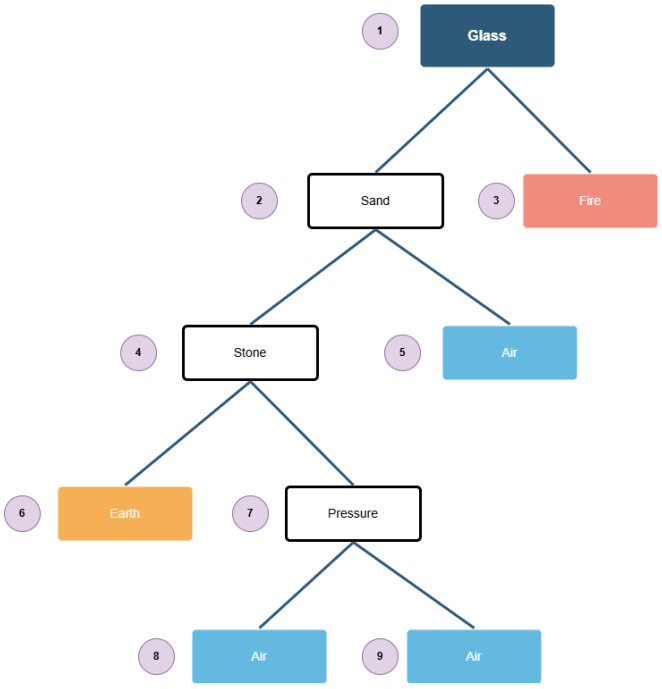
Gambar 5. Contoh Pohon DFS “Metal”

Iterasi	V	Q	dikunjungi

			1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
inisialisasi	1	{1}	**T**	F	F	F	F	F	F	F	F	F	F
1	1	{2,3}	**T**	**T**	**T**	F	F	F	F	F	F	F	F
2	2	{3, 4 ,5 }	**T**	**T**	**T**	**T**	**T**	**T**	F	F	F	F	F
3	3	{4, 5 , 8, 9}	**T**	**T**	**T**	**T**	**T**	F	F	**T**	**T**	F	F
4	4	{5, 8, 9}	**T**	**T**	**T**	**T**	**T**	F	F	**T**	**T**	F	F
5	5	{8,9,6,7 }	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	F	F
6	8	{9,6,7}	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	F	F
7	9	{6,7,10, 11}	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**
8	6	{7, 10, 11}	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**
9	7	{10, 11}	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**
10	10	{11}	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**
11	11	{0}	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**

Urutan simpul yang dikunjungi: 1, 2, 3, 4, 5, 8, 9, 6, 7, 10, 11

3.4.2 Depth-First Search (DFS)



Gambar 6. Contoh Pohon DFS "Glass"

Untuk memudahkan pemahaman mengenai pendekatan DFS yang dilakukan, maka diambil satu contoh kasus elemen target, yaitu Glass. Proses yang terjadi pada pembentukan tree elemen tersebut dapat dituliskan:

0: DFS (*Glass*): v=1; dikunjungi[1] = true
1: ├─ DFS (*Sand*): v=2; dikunjungi[2] = true
2: │ ├─ DFS (*Stone*): v=4; dikunjungi[4] = true
3: │ │ ├─ DFS (*Earth*): v=6; dikunjungi[6] = true; Base Element, valid
4: │ │ └─ Setelah Earth selesai → DFS (*Pressure*): v=7; dikunjungi[7] = true
5: │ │ ├─ DFS (*Air*): v=8; dikunjungi[8] = true; Base Element, valid
6: │ │ └─ Setelah *Air* selesai → DFS (*Air*): v=; dikunjungi[9] = true, Base element, valid
7: │ └─ Setelah Stone selesai → DFS (*Air*): v=; dikunjungi[5] = true, Base element, valid
8: └─ Setelah Sand selesai → DFS (*Fire*): v=; dikunjungi[3] = true, Base element, valid

Urutan simpul yang dikunjungi: 1,2,4,6,7,8,9,5,3

Terdapat beberapa informasi:

1. Suatu subtree dikatakan valid jika node leafnya merupakan base element.
2. Pemrosesan dilanjutkan (true) jika tier node yang dikunjungi < tier node parent.

3. Pemrosesan dilanjutkan jika node tidak base element dan masih terdapat elemen pembentuknya.

Adapun contoh bila terjadi kasus backtrack, seperti pada pemrosesan *Pressure*. Dapat dipilih:

1. *Air* + *Air*, hal ini akan membuat subtree valid
2. *Air* + *Atmosphere*, karena tier elemen *Atmosphere* (4) ≥ *Pressure* (1) maka langkah ini tidak valid sehingga dilakukan backtrack karena map visited pada *Pressure* bernilai false setelah keluar dari fungsi tersebut. Kemudian dicoba kemungkinan pembuatan elemen *Pressure* lainnya.

BAB 4
IMPLEMENTASI DAN PENGUJIAN

4.1. Implementasi Algoritma BFS

4.1.1. Struktur Data

Nama Variabel	Struktur Data	Penjelasan
MetricsResult	struct	Struktur data yang menyimpan metrik hasil pemrosesan BFS, terdiri dari NodesVisited (int64), Duration (int64 dalam milidetik), dan DurationHuman (string berupa durasi terbaca manusia).
Queue	struct []string	Struktur data antrian untuk mendukung traversal BFS, memiliki fungsi Enqueue, Dequeue, dan IsEmpty. Digunakan untuk menyimpan urutan elemen yang akan dieksplorasi.
nodesVisited	int64	Variabel penghitung jumlah node yang telah dikunjungi selama proses BFS.
recipes	[]Element	Menyimpan hasil resep (kombinasi elemen) yang ditemukan dari hasil BFS.

visited	map[string]bool	Digunakan untuk menandai elemen yang sudah dikunjungi selama BFS agar tidak diproses berulang.
recipeMap	map[string][]Element	Menyimpan seluruh resep elemen berdasarkan elemen root-nya. Setiap elemen dapat memiliki lebih dari satu resep.
resultTrees	[]Tree	Digunakan untuk menyimpan hasil rekonstruksi pohon dari resep-resep yang ditemukan.

4.1.2. Fungsi dan Prosedur

1. Fungsi MultipleRecipe

Fungsi ini sebagai convenience method yang menjadi perantara untuk memanggil fungsi lain agar lebih fleksibel terhadap proses penamaan fungsi.

```
func MultipleRecipe(name string, recipeMap
map[string][]Element.Element, count int) ([]Element.Tree,
MetricsResult) {
    return MultipleRecipeConcurrent(name, recipeMap, count)
}
```

2. Fungsi MultipleRecipeConcurrent

Fungsi ini digunakan untuk melakukan perhitungan node pada tree yang sudah dibangun, pengecekan kondisi awal apakah elemen yang dicari adalah base element (sehingga mempercepat proses), dan pemanggilan fungsi untuk melakukan pemrosesan tree dengan pendekatan Breadth-First Search (BFS). Pada fungsi ini diinisiasi dengan nilai variabel nodeVisited = 0 dan handle case insensitive pada nama elemen.

```
func MultipleRecipeConcurrent(name string, recipeMap
map[string][]Element.Element, count int) ([]Element.Tree,
MetricsResult) {
    startTime := time.Now()

    name = strings.ToLower(name)
    var trees []Element.Tree
    var nodesVisited int64

    if Element.IsBaseComponent(name) {
```

```
        trees = []Element.Tree{
            {
                Root: Element.Element{
                    Root:  name,
                    Left:  "",
                    Right: "",
                    Tier:  "0",
                },
                Children: nil,
            },
        }
        nodesVisited = 1
    } else {
        trees, nodesVisited = buildTreesBFS(name, recipeMap,
count)
    }
    if len(trees) > count {
        trees = trees[:count]
    }
    duration := time.Since(startTime)
    metrics := MetricsResult{
        NodesVisited:  nodesVisited,
        Duration:      duration.Milliseconds(),
        DurationHuman: duration.String(),
    }
    return trees, metrics
```

3. Fungsi buildTreesBFS

Fungsi ini digunakan untuk membangun semua pohon (tree) dari sebuah root menggunakan algoritma BFS. Fungsi ini memanfaatkan goroutine untuk mempercepat proses pembangunan pohon secara paralel dan menghitung jumlah node yang dikunjungi.

```
func buildTreesBFS(root string, recipeMap
map[string][]Element.Element, limit int) ([]Element.Tree, int64)
{
    if Element.IsBaseComponent(root) {
        return []Element.Tree{
            {
                Root: Element.Element{
                    Root:  root,
                    Left:  "",
                    Right: "",
                    Tier:  "0",
                },
                Children: nil,
            },
        }, 1
    }
```

```
    var nodesVisited int64 = 0
    var resultTrees []Element.Tree
    recipes, visitedCount := findRecipesBFS(root, recipeMap,
math.MaxInt32, limit*2)
    nodesVisited += visitedCount

    var mu sync.Mutex
    var wg sync.WaitGroup
    treeChan := make(chan []Element.Tree, len(recipes))

    for _, recipe := range recipes {
        if strings.ToLower(recipe.Root) != strings.ToLower(root)
{
            continue
        }
        wg.Add(1)
        go func(r Element.Element) {
            defer wg.Done()
            visited := make(map[string]bool)
            tierInt := Element.ParseTier(r.Tier)
            var localVisited int64 = 0
            trees := buildAllTreesFromRecipe(r, recipeMap,
visited, tierInt, limit, &localVisited)

            mu.Lock()
            nodesVisited += localVisited
            mu.Unlock()

            treeChan <- trees
        }(recipe)
    }
    wg.Wait()
    close(treeChan)

    for trees := range treeChan {
        for _, tree := range trees {
            if len(resultTrees) >= limit {
                break
            }
            resultTrees = append(resultTrees, tree)
        }
        if len(resultTrees) >= limit {
            break
        }
    }
    return resultTrees, nodesVisited
}
```

4. Fungsi findRecipesBFS

Fungsi ini melakukan pencarian semua resep dengan pendekatan BFS. Digunakan untuk mengumpulkan semua kombinasi resep dari suatu elemen root sampai batas tertentu (limit) dan tierLimit.

```
func findRecipesBFS(root string, recipeMap
map[string][]Element.Element, tierLimit int, limit int)
([]Element.Element, int64) {
    var nodesVisited int64 = 0
    var recipes []Element.Element

    if Element.IsBaseComponent(root) || root == "time" {
        return recipes, nodesVisited
    }

    queue := &Queue{}
    visited := make(map[string]bool)
    queue.Enqueue(root)

    for !queue.IsEmpty() && len(recipes) < limit {
        current := queue.Dequeue()
        current = strings.ToLower(current)
        nodesVisited++

        if visited[current] {
            continue
        }
        visited[current] = true

        currentRecipes, exists := recipeMap[current]
        if !exists {
            continue
        }

        for _, recipe := range currentRecipes {
            recipeInt := Element.ParseTier(recipe.Tier)
            if recipeInt < tierLimit {
                recipes = append(recipes, recipe)

                left := strings.ToLower(recipe.Left)
                right := strings.ToLower(recipe.Right)

                if !visited[left] {
                    queue.Enqueue(left)
                }
                if !visited[right] {
                    queue.Enqueue(right)
                }
            }
        }

        if len(recipes) >= limit {
            recipes = recipes[:limit]
            break
        }
    }

    return recipes, nodesVisited
}
```

5. Fungsi buildAllTreesFromRecipe

Fungsi rekursif ini membangun semua kemungkinan tree dari sebuah resep, mengeksplorasi anak kiri dan kanan, serta menangani komponen dasar. Juga mencatat jumlah node yang dikunjungi.

```
func buildAllTreesFromRecipe(recipe Element.Element, recipeMap
map[string][]Element.Element, visited map[string]bool, tierLimit
int, limit int, nodesVisited *int64) []Element.Tree {
    *nodesVisited++

    newVisited := cloneMap(visited)
    newVisited[strings.ToLower(recipe.Root)] = true

    var resultTrees []Element.Tree

    // Kiri
    left := strings.ToLower(recipe.Left)
    var leftTrees []Element.Tree
    if Element.IsBaseComponent(left) {
        *nodesVisited++
        leftTrees = append(leftTrees, Element.Tree{
            Root: Element.Element{Root: left, Tier: "0"},
            Children: nil,
        })
    } else if !newVisited[left] {
        if leftRecipes, exists := recipeMap[left]; exists {
            for _, leftRecipe := range leftRecipes {
                if Element.ParseTier(leftRecipe.Tier) < tierLimit
{
                    leftSubtrees :=
buildAllTreesFromRecipe(leftRecipe, recipeMap, newVisited,
Element.ParseTier(leftRecipe.Tier), limit, nodesVisited)
                    leftTrees = append(leftTrees,
leftSubtrees...)
                    if len(leftTrees) >= limit {
                        break
                    }
                }
            }
        }
    }

    if len(leftTrees) == 0 {
        return []Element.Tree{}
    }

    // Kanan
    right := strings.ToLower(recipe.Right)
    var rightTrees []Element.Tree
    if Element.IsBaseComponent(right) {
        *nodesVisited++
        rightTrees = append(rightTrees, Element.Tree{
            Root: Element.Element{Root: right, Tier: "0"},
```

```
            Children: nil,
        })
    } else if !newVisited[right] {
        if rightRecipes, exists := recipeMap[right]; exists {
            for _, rightRecipe := range rightRecipes {
                if Element.ParseTier(rightRecipe.Tier) <
tierLimit {
                    rightSubtrees :=
buildAllTreesFromRecipe(rightRecipe, recipeMap, newVisited,
Element.ParseTier(rightRecipe.Tier), limit, nodesVisited)
                    rightTrees = append(rightTrees,
rightSubtrees...)
                    if len(rightTrees) >= limit {
                        break
                    }
                }
            }
        }
    }

    if len(rightTrees) == 0 {
        return []Element.Tree{}
    }

    for _, lt := range leftTrees {
        for _, rt := range rightTrees {
            resultTrees = append(resultTrees, Element.Tree{
                Root:     recipe,
                Children: []Element.Tree{lt, rt},
            })
            if len(resultTrees) >= limit {
                return resultTrees
            }
        }
    }

    return resultTrees
}
```

6. Fungsi PrintTree

Fungsi ini digunakan untuk mencetak pohon dalam format terindentasi, memudahkan proses debugging dan visualisasi hasil tree.

```
func PrintTree(t Element.Tree, indent string) {
    fmt.Printf("%s%s (Tier: %s)\n", indent, t.Root.Root,
t.Root.Tier)
    for _, child := range t.Children {
        PrintTree(child, indent + "  ")
    }
}
```

4.2. Implementasi Algoritma DFS

4.2.1. Struktur Data

Nama Variabel	Struktur Data	Penjelasan
BaseComponents	map[string]bool	Struktur data yang dibentuk untuk memetakan base element dan nilai boolean, yaitu: "air": true "earth": true "fire": true "water": true "time": false
Element	struct	Struktur data yang dibentuk dengan menggunakan struct dan terdiri dari beberapa elemen antara lain Root, Left, Right, dan Tier yang bertipe string. Isi nilai dari elemen ini akan disesuaikan dengan pengambilan dari file .json yang merupakan hasil *scraping*.
allElements	[]Element	Digunakan untuk menyimpan hasil seluruh informasi dari file .json.
MetricResult	struct	Struktur data yang dibentuk dengan menggunakan struct dan terdiri dari beberapa elemen antara lain NodesVisited, Duration yang berisi type int64 dan DurationHuman yang berupa string. Elemen ini digunakan sebagai nilai return pada hasil pemrosesan DFS.
trees	[]Tree	Digunakan untuk menyimpan hasil tree dan sebagai return dari DFS.
visited	map[string]bool	Struktur data yang dibentuk untuk memetakan element dan nilai boolean, yaitu akan bernilai true jika sudah dikunjungi dan false jika belum.
recipeMap	map[string][]Element	Digunakan untuk menyimpan seluruh elemen dengan root tertentu.

4.2.2. Fungsi dan Prosedur

1. Fungsi MultipleRecipe

Fungsi ini sebagai convenience method yang menjadi perantara untuk memanggil fungsi lain agar lebih fleksibel terhadap proses penamaan fungsi.

```
func MultipleRecipe(name string, recipeMap
map[string][]Element.Element, count int) ([]Element.Tree,
MetricsResult) {
	return MultipleRecipeConcurrent(name, recipeMap, count)
}
```

2. Fungsi MultipleRecipeConcurrent

Fungsi ini digunakan untuk melakukan perhitungan node pada tree yang sudah dibangun, pengecekan kondisi awal apakah elemen yang dicari adalah base element (sehingga mempercepat proses), dan pemanggilan fungsi untuk melakukan pemrosesan tree. Pada fungsi ini diinisiasi dengan nilai variabel nodeVisited = 0 dan handle case insensitive pada nama element.

```
// Perhitungan node dan pengecekan kondisi tree yang dapat
dibangun (base/not)
func MultipleRecipeConcurrent(name string, recipeMap
map[string][]Element.Element, count int) ([]Element.Tree,
MetricsResult) {
	startTime := time.Now()
	var nodesVisited int64 = 0
	var baseComp bool
	name = strings.ToLower(name)
	var trees []Element.Tree
	if Element.IsBaseComponent(name) {
		baseComp = true
		trees = []Element.Tree{
			{
				Root: Element.Element{
					Root:  name,
					Left:  "",
					Right: "",
					Tier:  "0",
				},
				Children: nil,
			},
		}
	} else {
		baseComp = false
		trees = BuildTrees(name, recipeMap,
map[string]bool{}, math.MaxInt32, count)
	}

	if len(trees) > count {
		trees = trees[:count]
	}

	for _, tree := range trees {
```

```
            nodesVisited += CountNodes(tree)
      }

      if baseComp {
            duration := time.Since(startTime)
            metrics := MetricsResult{
                  NodesVisited:  1,
                  Duration:      duration.Milliseconds(),
                  DurationHuman: duration.String()}
            return trees, metrics
      } else {
            duration := time.Since(startTime)
            metrics := MetricsResult{
                  NodesVisited:  nodesVisited,
                  Duration:      duration.Milliseconds(),
                  DurationHuman: duration.String()}
            return trees, metrics
      }
}
```

3. Fungsi BuildTrees

Fungsi ini akan melakukan pemrosesan pencarian tree yang valid secara rekursif. Base condition pada fungsi ini adalah jika root yang di-*visit* adalah base element, sudah pernah di-visit, tier element ≥ parent, atau tidak ditemukan recipe yang memenuhi untuk membuat elemen tersebut pada RecipeMap.

Pemrosesan dilakukan dengan menggunakan 2 goroutine setiap prosesnya untuk membedakan proses pembentukan subtree sebelah kiri dan kanan. Pemrosesan DFS akan dimulai terlebih dahulu di sebelah kiri secara rekursif dan jika sudah selesai maka dilakukan pemrosesan untuk subtree kanan. Hal ini memungkinan paralelisme pada subtree kiri.

Di tiap pemrosesan dalam pembangunan kombinasi subtree kiri dan subtree kanan menjadi satu tree yang valid, akan disesuaikan dengan banyaknya subtree kiri yang berhasil dibentuk (jumlah pembuatan subtree kanan dibatasi hingga banyaknya pohon yang diinginkan / jumlah subtree kiri). Hal ini dimaksudkan untuk optimalisasi pembuatan tree sesuai kebutuhan saja. Sehingga tidak perlu mengeksplorasi seluruh subtree yang valid. Tetapi, di akhir tetap dilakukan pengecekan jumlah tree yang akan di-*return* sebagai pengecekan yang lebih pasti karena proses berjalan secara paralel (*race condition*).

```
// Cari Tree yang Valid
func BuildTrees(root string, recipeMap
```

```
map[string][]Element.Element, visited map[string]bool, tierLimit
int, limit int) []Element.Tree {
      if Element.IsBaseComponent(root) {
            return []Element.Tree{
                  {
                        Root: Element.Element{
                              Root:  root,
                              Left:  "",
                              Right: "",
                              Tier:  "0",
                        },
                        Children: nil,
                  },
            }
      }

      if visited[root] {
            return nil
      }

      recipes, exists := recipeMap[strings.ToLower(root)]
      if !exists || root == "time" {
            return nil
      }

      var result []Element.Tree
      visited[root] = true
      defer func() { visited[root] = false }()

      for _, recipe := range recipes {
            tierInt := Element.ParseTier(recipe.Tier)
            if tierInt >= tierLimit {
                  continue
            }

            left := strings.ToLower(recipe.Left)
            right := strings.ToLower(recipe.Right)

            var leftTrees, rightTrees []Element.Tree
            var wg sync.WaitGroup

            leftChan := make(chan []Element.Tree, 1)
            rightChan := make(chan []Element.Tree, 1)

            wg.Add(2)

            // Proses kiri secara paralel
            go func() {
                  defer wg.Done()
                  leftChan <- BuildTrees(left, recipeMap,
CloneVisited(visited), tierInt, limit)
            }()

            // Proses kanan setelah dapat hasil subtree kiri
            go func() {
                  defer wg.Done()
```

```
                  leftResult := <-leftChan
                  leftTrees = leftResult
                  if len(leftResult) == 0 {
                        rightChan <- nil
                        return
                  }
                  rightLimit := int(math.Ceil(float64(limit) /
float64(len(leftResult))))
                  rightChan <- BuildTrees(right, recipeMap,
CloneVisited(visited), tierInt, rightLimit)
            }()

            wg.Wait()
            rightTrees = <-rightChan

            if len(leftTrees) == 0 || len(rightTrees) == 0 {
                  continue
            }

            for _, leftT := range leftTrees {
                  for _, rightT := range rightTrees {
                        tree := Element.Tree{
                              Root:     recipe,
                              Children: []Element.Tree{leftT,
rightT},
                        }
                        result = append(result, tree)
                        if len(result) >= limit {
                              return result
                        }
                  }
            }
      }

      if len(result) > limit {
            result = result[:limit]
      }
      return result
}
```

4. Fungsi CloneVisited

Fungsi ini digunakan untuk melakukan *cloning* pada struktur dapat map pada variabel visited. Hal ini dimaksudkan sehingga setiap element yang diproses pada fungsi BuildTrees akan memiliki mappingnya sendiri dan bisa memungkinkan proses backtracking.

```
// Untuk clone map visited
func CloneVisited(visited map[string]bool) map[string]bool {
      copy := make(map[string]bool)
      for k, v := range visited {
            copy[k] = v
      }
```

```
	return copy
}
```

5. Fungsi CountNodes

	Fungsi ini digunakan untuk menghitung jumlah node pada 1 tree dengan mengakses jumlah children yang dimiliki.

```
// Menghitung jumlah node pada 1 tree
func CountNodes(tree Element.Tree) int64 {
	count := int64(1)
	for _, child := range tree.Children {
		count += CountNodes(child)
	}
	return count
}
```

6. Fungsi BuildRecipeMap

	Fungsi ini digunakan untuk mengelompokkan informasi elemen berdasarkan nama root pada struct Element.

```
// Kelompokkan berdasarkan rootnya
func BuildRecipeMap(recipes []Element) map[string][]Element {
	recipeMap := make(map[string][]Element)
	for _, r := range recipes {
		recipeMap[strings.ToLower(r.Root)] =
append(recipeMap[strings.ToLower(r.Root)], r)
	}
	return recipeMap
}
```

7. Fungsi IsBaseComponent

	Fungsi ini digunakan untuk mengecek apakah elemen yang ingin dicek merupakan base element atau tidak.

```
// Cek apakah base component
func IsBaseComponent(item string) bool {
	return BaseComponents[strings.ToLower(item)]
}
```

8. Fungsi ParseTier

	Fungsi ini digunakan untuk mengubah tipe data dari tier yang berupa string menjadi integer. Hal ini dimaksudkan untuk memudahkan dalam pengecekan aturan valid pada pengaksesan suatu node.

```
// Ubah tier ke type integer
func ParseTier(tierStr string) int {
```

```
        var tier int
        fmt.Sscanf(tierStr, "%d", &tier)
        return tier
}
```

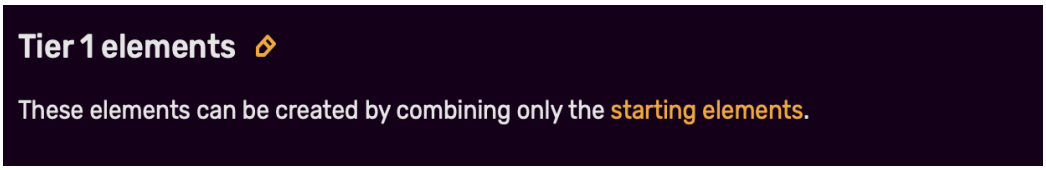
4.3. Implementasi Website

Program dibuat berbasis website yang dibagun melalui docker. Fitur-fitur yang terdapat pada website antara lain : scraping website untuk pengumpulan data, pencarian resep menggunakan metode BFS, pencarian resep menggunakan metode dfs, dan pembuatan graf pohon untuk visualisasi resep hasil pencarian resep.

4.3.1. Scraping Data

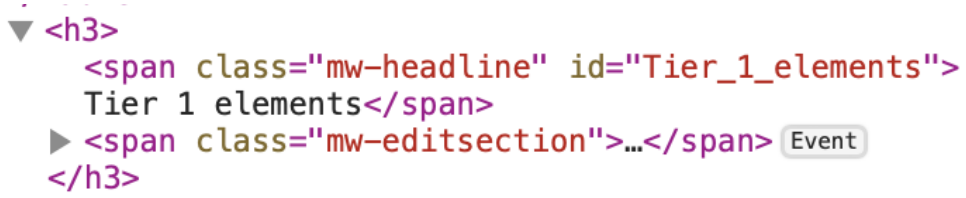
Scraping data merupakan suatu proses pengambilan data. Data yang diambil adalah resep berdasarkan website https://little-alchemy.fandom.com/wiki/Elements_(Little_Alchemy_2). Scraping dilakukan dengan mengambil response html pranala tersebut dan mengambil tag html yang dirasa perlu untuk kepentingan data. Data yang diambil merupakan data elemen yang berisi nama elemen (root) , elemen pembangun (*left* dan *right*) , dan *tier*. Hasil scrap disimpan dalam format JSON untuk memudahkan penggunaan pada website. Scrapping website diimplementasikan pada file scrapper.go .

4.3.1.1. Tier



Gambar 7. Potongan Screenshot Informasi Tier pada Web Little Alchemy Fandom

Informasi *tier* didapat melalui komponen tersebut, scraper akan mengambil isi dari html tag dan menyimpannya sebagai informasi tier. Tag yang diambil adalah tag berikut.



Gambar 8. Potongan Screenshot HTML Tag Tentang Informasi Tier

Untuk mengambil dan menyimpan data yang terdapat pada tag html tersebut. Digunakan regex dan kode seperti berikut akan mengambil tier apa saja yang terdapat pada website target.

```
tierTableRegex := regexp.MustCompile(
        `(?is)<span class="mw-headline"
id="(Tier_\d+)_elements">.*?</span>.*?(<table.*?>.*?</table>)`)
    tierTableMatches := tierTableRegex.FindAllStringSubmatch(html,
-1)
```

4.3.1.2. Element

Gambar 9. Potongan Screenshot Recipes Element

Data element didapat dari tabel tersebut, tabel memiliki dua kolom Element dan Recipes, Element merupakan data element yang menjadi parent dan recipe merupakan elemen pembangun.

```
<tr>
  <th style="width:200px;">Element </th>
  <th>Recipes </th>
</tr>
```

Gambar 10. Potongan Screenshot HTML Tag Tentang Recipes

Cuplikan tag html diatas merupakan penanda bahwa tabel memiliki 2 *column* yaitu Elemen dan Recipes. Data yang akan diambil adalah komponen *row* dibawahnya yaitu tag berikut.

Gambar 11. Potongan Screenshot HTML Tag Tentang Informasi Recipes

Tag tersebut berisi row data element pada contoh ini “Dust” serta resep pembangunnya. Untuk mengambil dan menyimpan data tersebut digunakan regex dan kode Go sebagai berikut.

```
func extractKomposers(td string) [][]string {
    liRegex := regexp.MustCompile(`(?is)<li[^>]*>.*?</li>`)
    liMatches := liRegex.FindAllString(td, -1)
    var pairs [][]string
    for _, li := range liMatches {
        spanRegex :=
regexp.MustCompile(`(?is)<span[^>]*>.*?</span>`)
        cleanLi := spanRegex.ReplaceAllString(li, "")
        aRegex := regexp.MustCompile(`(?is)<a[^>]*>(.*?)</a>`)
        aMatches := aRegex.FindAllStringSubmatch(cleanLi, -1)
        if len(aMatches) == 2 {
            left := cleanText(aMatches[0][1])
            right := cleanText(aMatches[1][1])
            pairs = append(pairs, []string{left, right})
        }
    }
    return pairs
}
```

```
func extractElementsFromTable(tableHtml string, tier string)
[]Element.Element {
    var elements []Element.Element
    rowRegex := regexp.MustCompile(`(?is)<tr.*?>.*?</tr>`)
    rows := rowRegex.FindAllString(tableHtml, -1)
    for _, row := range rows {
        tdRegex := regexp.MustCompile(`(?is)<td.*?>.*?</td>`)
        tds := tdRegex.FindAllString(row, -1)
        if len(tds) < 1 {
            continue
        }
        elementRegex := regexp.MustCompile(`(?is)<a
href="/wiki/[^"]*" title="[^"]*">(.*?)</a>`)
        elementMatch := elementRegex.FindStringSubmatch(tds[0])
        root := "UNKNOWN"
        if len(elementMatch) > 1 {
            root = cleanText(elementMatch[1])
        }
        if len(tds) >= 2 {
            komposers := extractKomposers(tds[1])
              if len(komposers) > 0 {
                for _, pair := range komposers {
                    elements = append(elements, Element.Element{
                        Root:  root,
                        Left:  pair[0],
                        Right: pair[1],
                        Tier:  tier,
                    })
                }
                continue
            }

            elements = append(elements, Element.Element{
                Root:  root,
                Left:  "",
                Right: "",
                Tier:  "0",
            })
            continue
        }
            elements = append(elements, Element.Element{
            Root:  root,
            Left:  "",
            Right: "",
            Tier:  tier,
        })
    }
    return elements
```

```
}
```

```
    for _, match := range tierTableMatches {
        tierRaw := match[1]
        tier := strings.Split(tierRaw, "_")[1]
        tableHtml := match[2]
        elements := extractElementsFromTable(tableHtml, tier)
        allElements = append(allElements, elements...)
    }
```

Fungsi extractComposer memiliki tugas untuk mengambil data resep pembangun. Fungsi extractElemenFromTable memiliki fungsi untuk mengambil data pada tabel secara keseluruhan dan membentuknya dalam json. Cuplikan fungsi terakhir merupakan bagian dari fungsi scraper yang berfungsi untuk menyatukan atau menghubungkan antara tier dengan elemen yang sesuai.

Hasil dari scraper merupakan file dalam format JSON seperti contoh berikut :


```
[
 . . .
  {
    "root": "Air",
    "Left": "Fire",
    "Right": "Mist",
    "Tier": "0"
  },
  {
    "root": "Earth",
    "Left": "",
    "Right": "",
    "Tier": "0"
  },
  {
    "root": "Fire",
    "Left": "Fire",
    "Right": "Alcohol",
    "Tier": "0"
  },
 . . .
]
```

4.3.2. Handler

Handler digunakan sebagai perantara antara algoritma query dengan dunia di luar program. Handler melakukan konversi tipe data Tree[] ke dalam format JSON. Terdapat tiga handler pada program ini yaitu BFSHandler, DFSHandler, dan ScrapperHandler.

```
func BFSHandler(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query().Get("element")
    countStr := r.URL.Query().Get("count")
    count, err := strconv.Atoi(countStr)
    if err != nil {
        fmt.Println("Conversion error:", err)
    } else {
        fmt.Println("Converted int:", count)
    }
    recipeMap := Element.BuildRecipeMap(Element.GetAllElement())
    result, info := bfs.MultipleRecipe(name, recipeMap, count)
    Element.AreAllTreesUnique(result)
    response := []interface{}{info, result}
    w.Header().Set("Content-Type", "application/json")
    if err := json.NewEncoder(w).Encode(response); err != nil {
        http.Error(w, "Error encoding JSON",
http.StatusInternalServerError)
        fmt.Println("JSON encode error:", err)
    }
}
```

```
func DFSHandler(w http.ResponseWriter, r *http.Request) {
    name := r.URL.Query().Get("element")
    countStr := r.URL.Query().Get("count")
    count, err := strconv.Atoi(countStr)
    if err != nil {
        fmt.Println("Conversion error:", err)
    } else {
        fmt.Println("Converted int:", count)
    }
    recipeMap := Element.BuildRecipeMap(Element.GetAllElement())
    result, info := dfs.MultipleRecipe(name, recipeMap, count)
    Element.AreAllTreesUnique(result)
    response := []interface{}{info, result}
    w.Header().Set("Content-Type", "application/json")
    if err := json.NewEncoder(w).Encode(response); err != nil {
        http.Error(w, "Error encoding JSON",
http.StatusInternalServerError)
        fmt.Println("JSON encode error:", err)
    }
}
```

```
func ScrapHandler(w http.ResponseWriter, r *http.Request) {
    scrapper.Scrapper()
    fmt.Fprintln(w, "Scraping selesai.")
    Element.LoadElementsFromFile("output.json")
    fmt.Fprintln(w, "Load selesai")
}
```

4.3.3. Main

Main merupakan file yang mengatur http request pada program back-end. Main mengatur rute apa saja yang tersedia dan melakukan pemetaan kepada handler yang sesuai.

```
func main() {
    http.HandleFunc("/Scrap", enableCORS(handler.ScrapHandler))
    http.HandleFunc("/BFS", enableCORS(handler.BFSHandler))
    http.HandleFunc("/DFS", enableCORS(handler.DFSHandler))
    fmt.Println("Server is running on http://localhost:8080")
    http.ListenAndServe(":8080", nil)
}
```

4.4. Tata Cara Penggunaan Program

4.4.1. Persiapan Program

Sebelum menggunakan program harus dipastikan program sudah tersedia pada perangkat. Program dapat di-*clone* melalui pranala pada bagian lampiran. Pastikan docker sudah terinstall dan siap digunakan. Jika kebutuhan awal sudah selesai, maka nyalakan container BE dan FE dengan perintah,

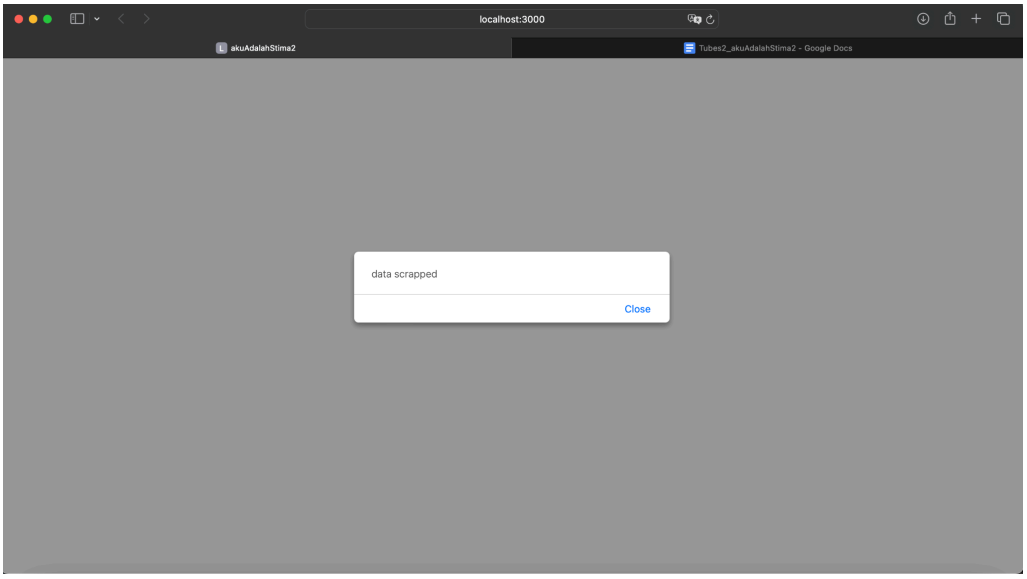
```
// untuk melakukan build
Docker-compose build

// menyalakan container
Docker-compose up

//mematikan container
Docker-compose down
```

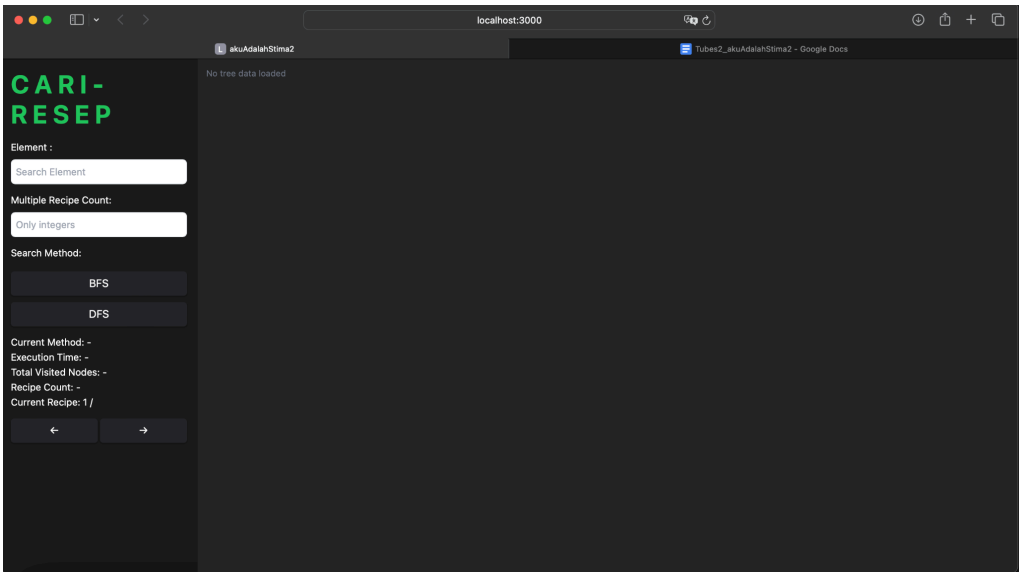
4.4.2. Inisiasi Awal Program

Setelah menyalakan container, masuk ke dalam jaringan localhost pada port 3000, lokasi dimana aplikasi *front-end* berjalan.



Gambar 12. Tampilan Awal Web

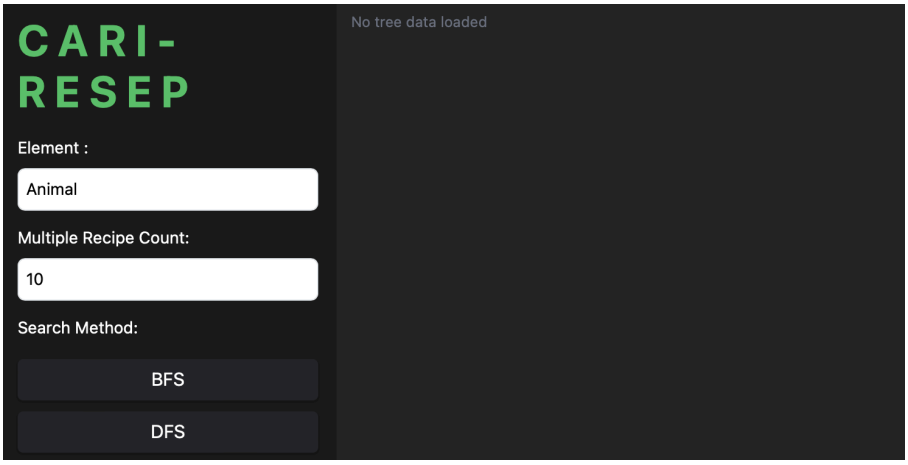
Program akan melakukan scraping data saat pertama kali dijalankan atau lebih tepatnya setiap kali ada request ke localhost:3000/ . Setelahnya, klik tombol “close” untuk masuk ke menu utama.



Gambar 12. Tampilan Web Keseluruhan

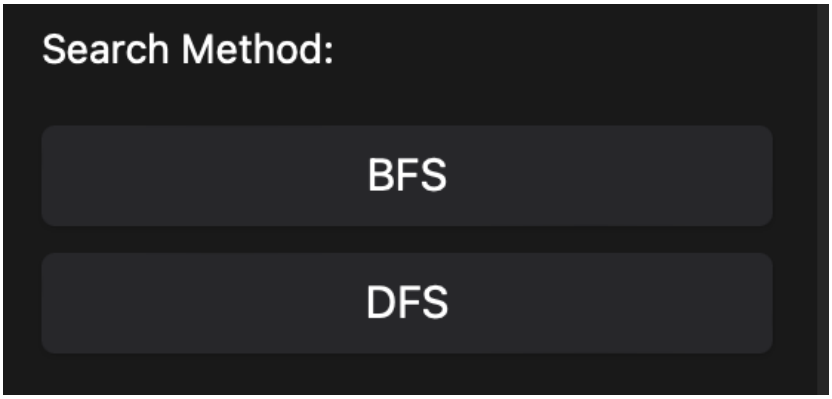
4.4.3. Pencarian Resep

Untuk mencari resep, pengguna harus mengisi nama resep yang ingin dicari pada inputbox “element” yang terletak di sidebar kemudian menuliskan jumlah resep yang ingin dicari.



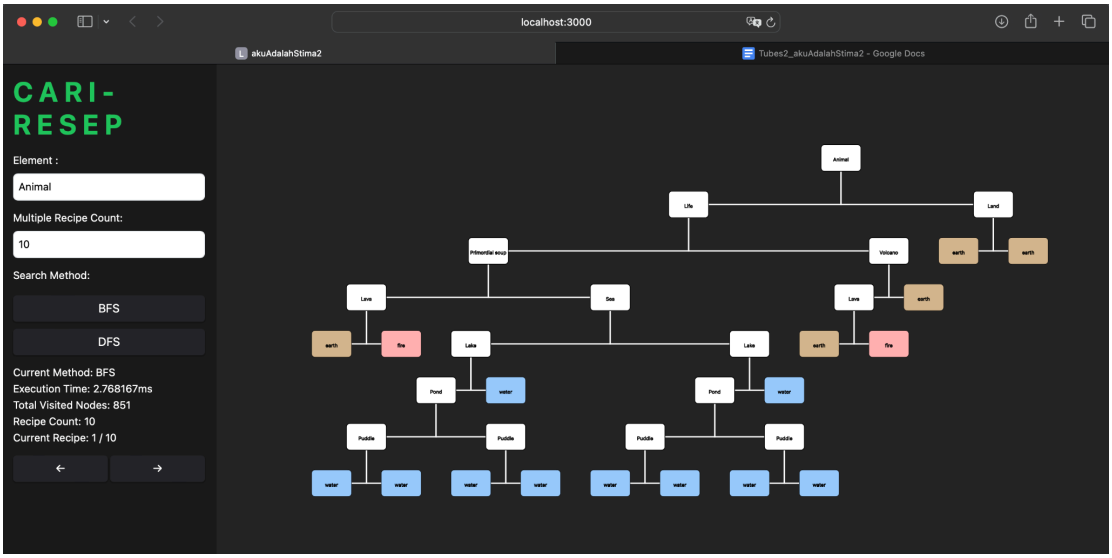
Gambar 13. Potongan Screenshot Website pada Bagian Input dan Output

Sebagai contoh akan dimasukkan nilai element yaitu “Animal” dan jumlah resep yang ingin dicari adalah 10. Setelah memasukkan data awal, pilih salah satu tombol “DFS” atau “BFS” untuk mencari resep target. Tombol “DFS” akan mencari resep dengan metode DFS sedangkan tombol “BFS” akan mencari resep dengan metode BFS.



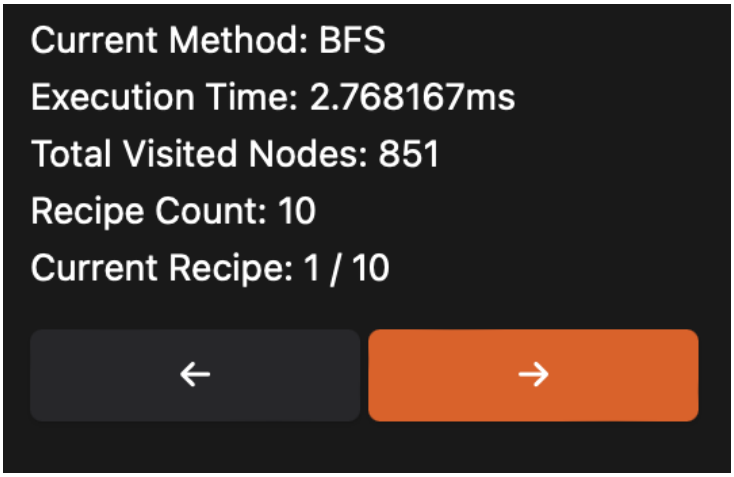
Gambar 14. Potongan Screenshot Pilihan Metode Pemrosesan Tree

Jika salah satu tombol tersebut di klik dan elemen serta jumlah resep yang dicari valid, maka akan muncul graf pohon resep pada screen utama. Namun jika input tidak valid atau resep tidak ditemukan maka tidak akan muncul apapun pada screen utama.



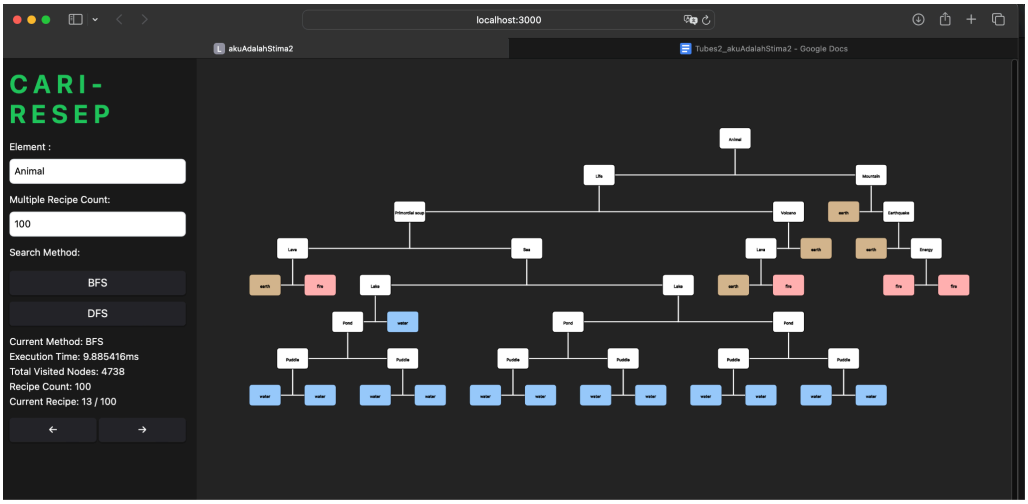
Gambar 15. Tampilan Hasil Proses Pencarian Resep dari ‘Animal’

Pada layar akan tertampil satu resep unik. Untuk melihat resep selanjutnya dapat ditekan tombol panah yang terletak pada *sidebar* di bawah informasi resep.



Gambar 16. Potongan Screenshot Informasi Hasil Pemrosesan

Jika tombol tersebut ditekan, tampilan resep akan berubah ke resep lain dari target namun berbeda dengan resep sebelumnya.



Gambar 17. Tampilan Tree Selanjutnya dari Recipe ‘Animal’

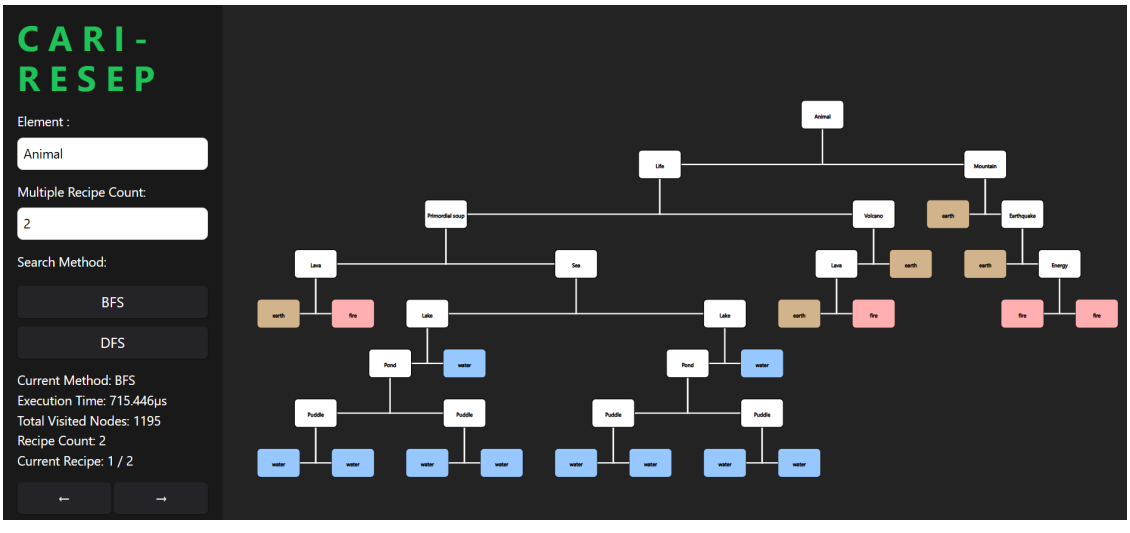
4.5. Hasil Pengujian

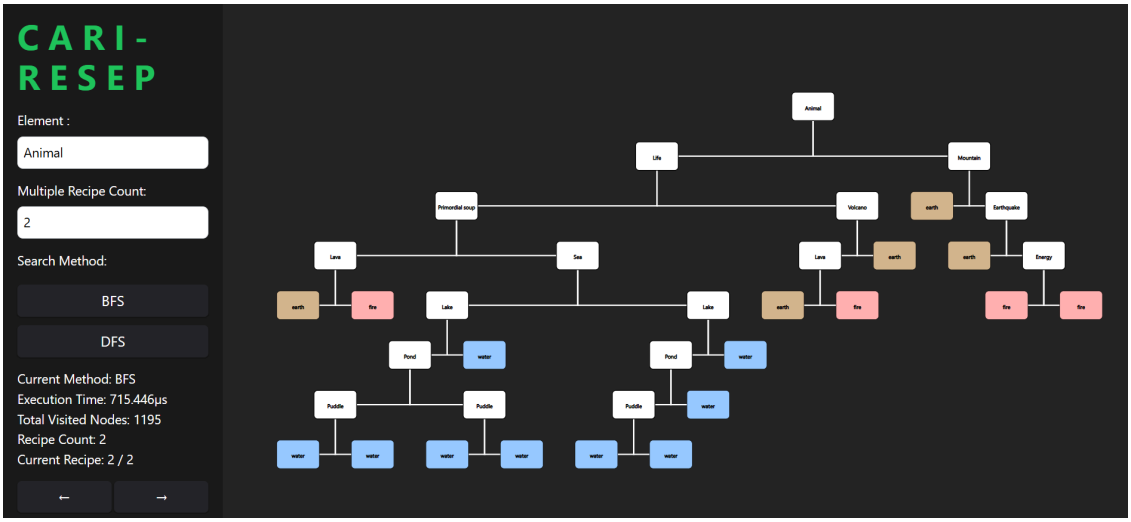
4.4.1. Test Case 1

Input : Animal

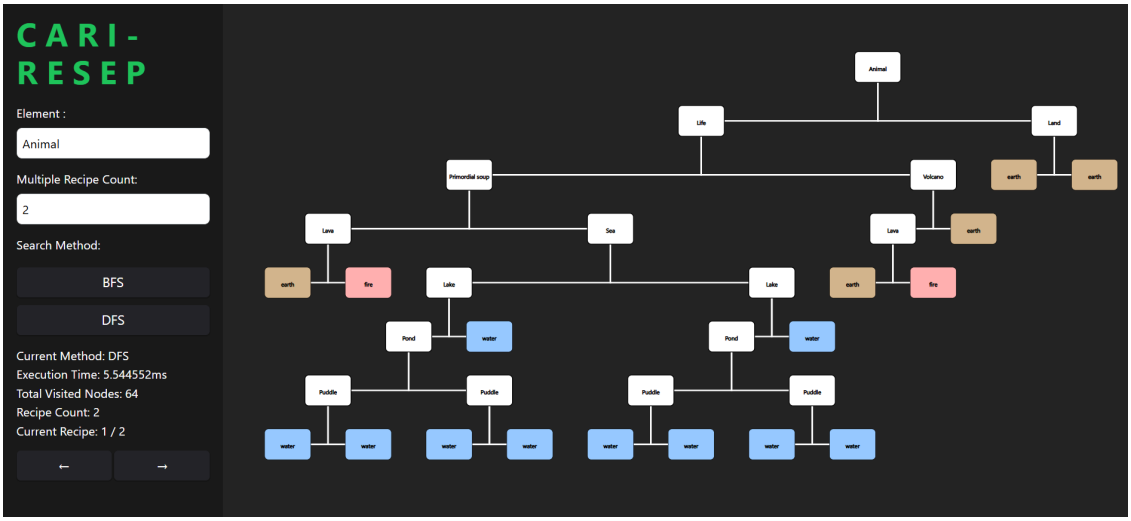
Multiple Recipe : 2

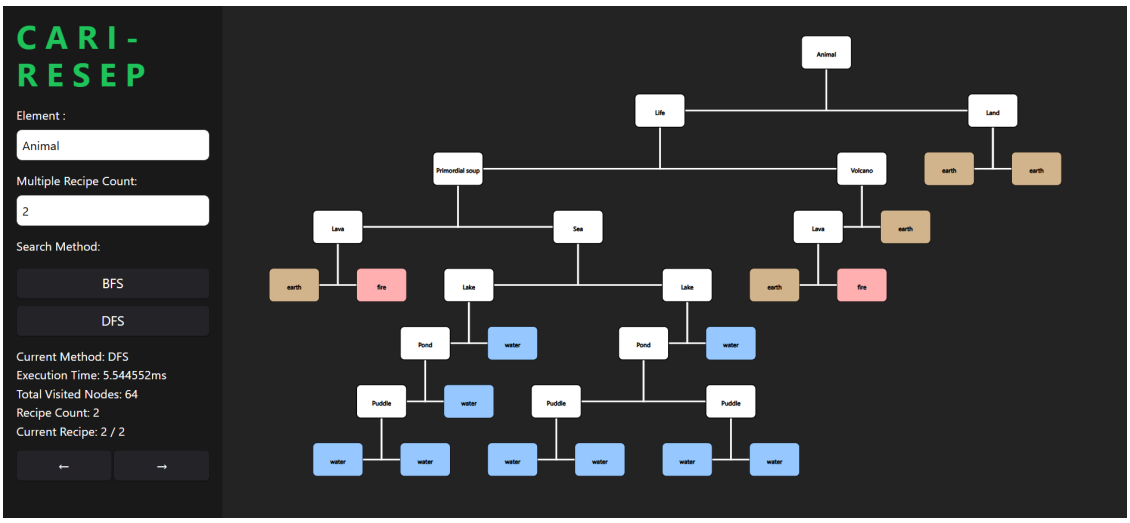
Test case ini bertujuan untuk menunjukkan bahwa proses pencarian BFS dan DFS dapat dilakukan dan valid.

Metode BFS



CARI-
RESEP
Element :
Animal
Multiple Recipe Count:
2
Search Method:
BFS
DFS
Current Method: BFS
Execution Time: 715.446µs
Total Visited Nodes: 1195
Recipe Count: 2
Current Recipe: 2 / 2
←
→

Metode DFS


CARI-
RESEP
Element :
Animal
Multiple Recipe Count:
2
Search Method:
BFS
DFS
Current Method: DFS
Execution Time: 5.544552ms
Total Visited Nodes: 64
Recipe Count: 2
Current Recipe: 1 / 2
←
→



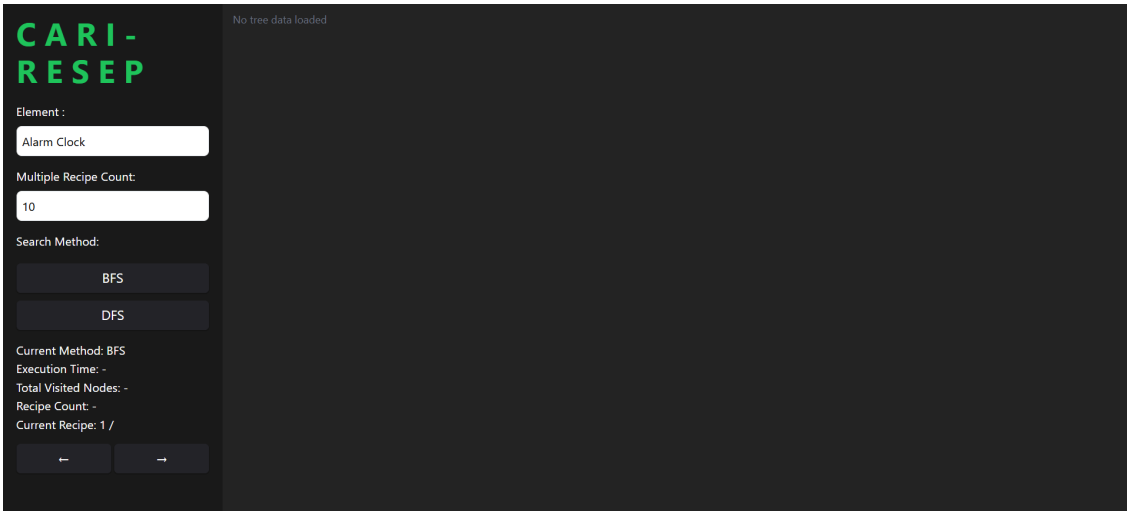
4.4.2. Test Case 2

Input : Alarm Clock

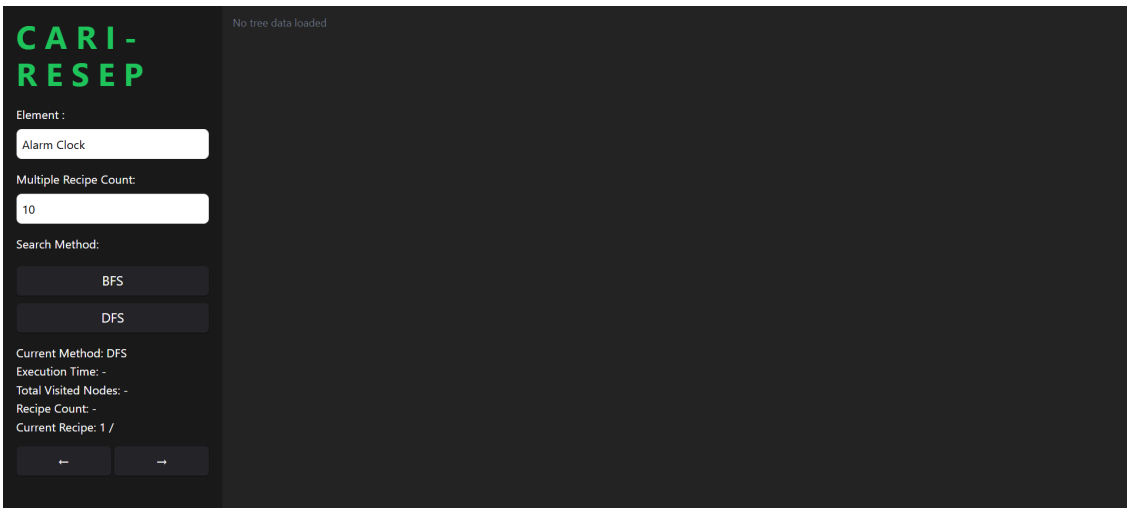
Multiple Recipe : 10

Test case ini bertujuan untuk menunjukkan bahwa proses pencarian BFS dan DFS tidak dapat menghasilkan tree yang valid karena elemen “Time” tidak termasuk menjadi base element.

Metode BFS



Metode DFS



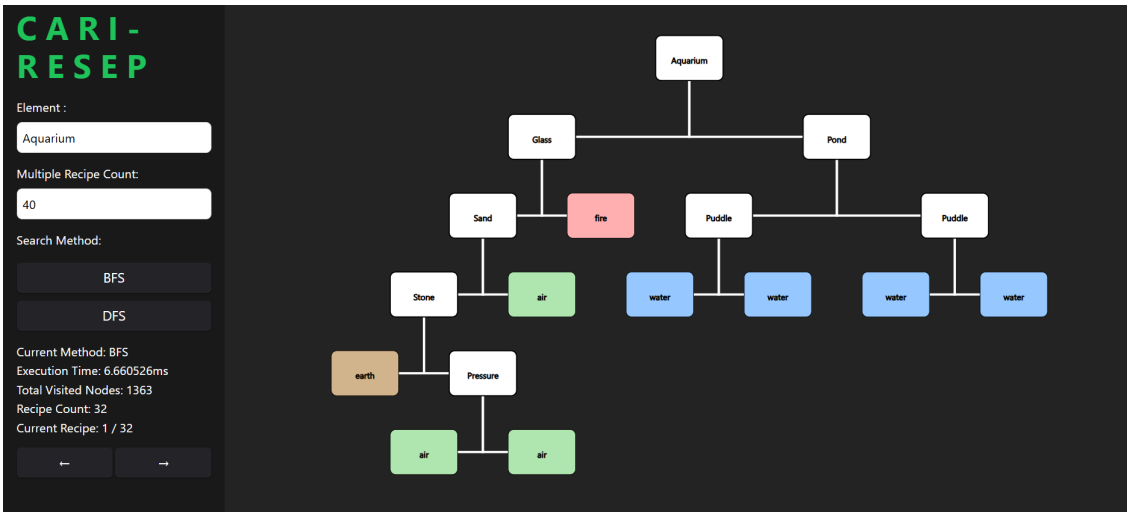
4.4.3. Test Case 3

Input : Aquarium

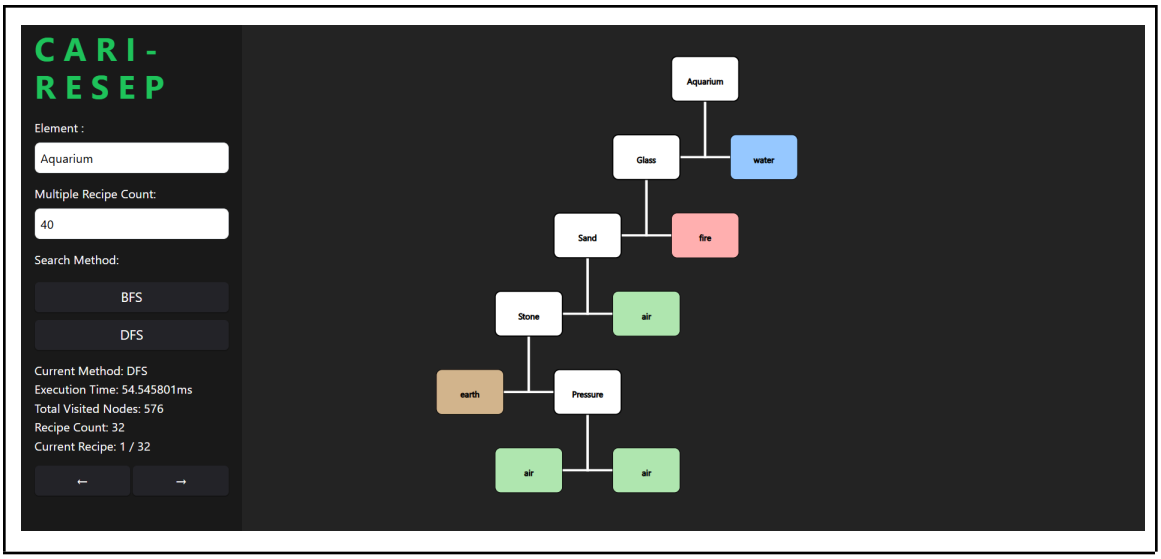
Multiple Recipe : 40

Test case ini bertujuan untuk menunjukkan bahwa proses pencarian BFS dan DFS terkadang menghasilkan tree valid yang terbatas (tidak dapat memenuhi jumlah recipe yang diinginkan user) bila informasi elemen memang tidak memungkinkan untuk menciptakan tree sebanyak inputan. Pada test case ini, hanya didapatkan 32 tree unik

Metode BFS



Metode DFS

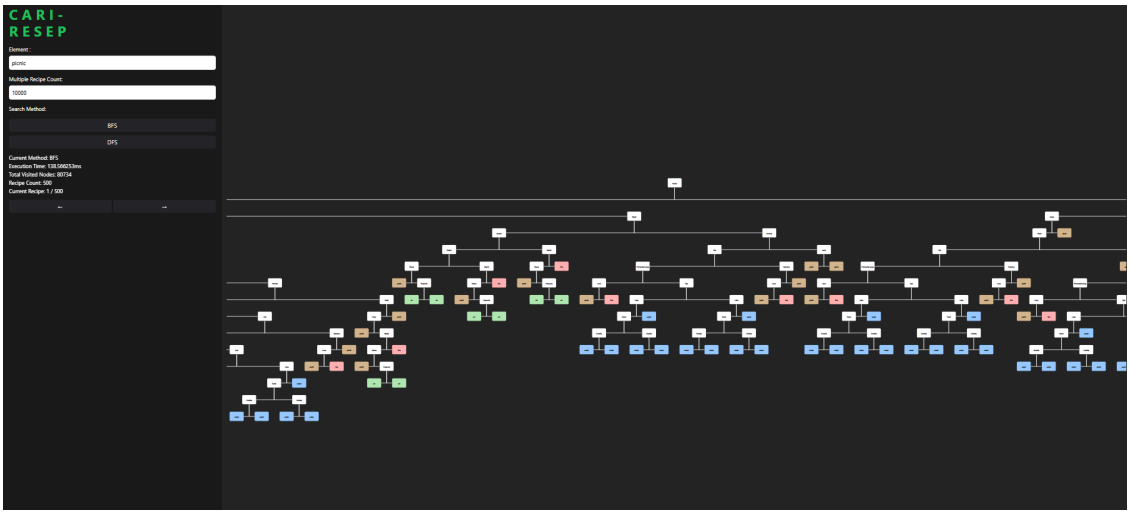


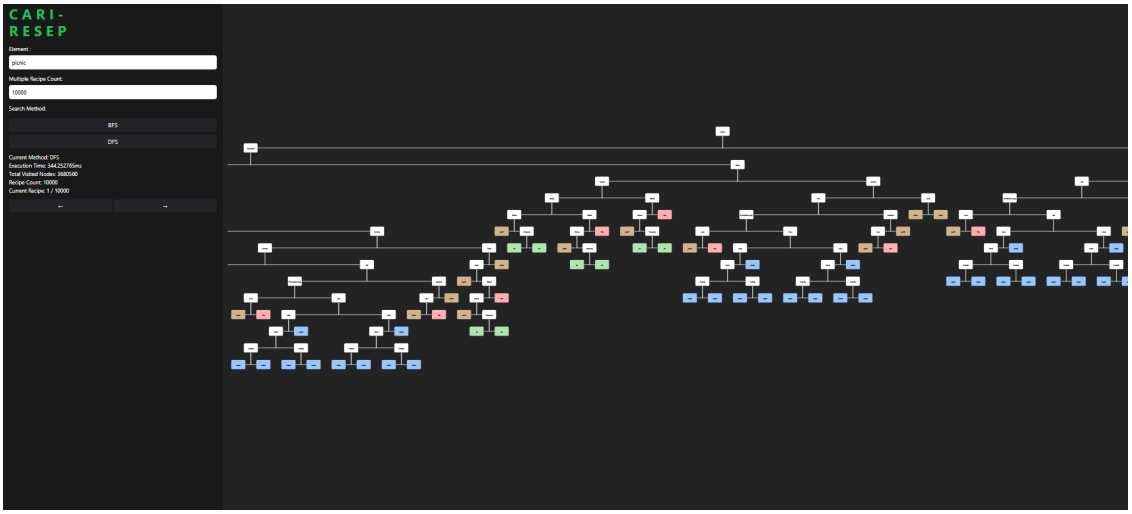
4.4.4. Test Case 4

Input : Picnic

Multiple Recipe : 10000

Test case ini bertujuan untuk menunjukkan bahwa proses pencarian BFS dan DFS dapat dilakukan secara efisien dan optimal dengan tier elemen yang tinggi juga.

Metode BFS

*Halaman tidak cukup untuk menampilkan keseluruhan visualisasi tree
Metode DFS



*Halaman tidak cukup untuk menampilkan keseluruhan visualisasi tree

4.5. Analisis Hasil Pengujian

Berdasarkan hasil pengujian yang dilakukan, terlihat bahwa hasil pohon (*tree*) dari pemrosesan menggunakan metode DFS dan BFS berbeda. Hal ini dapat diamati pada visualisasi, di mana DFS menghasilkan struktur pohon yang lebih dalam di sisi kiri, sedangkan BFS membentuk struktur yang lebih melebar. Perbedaan ini secara intuitif sesuai dengan karakteristik dasar dari algoritma DFS dan BFS.

Secara umum, waktu yang dibutuhkan untuk memproses dengan DFS maupun BFS relatif cepat dan tidak menunjukkan perbedaan yang signifikan. Salah satu penyebabnya adalah penggunaan *goroutine*, yaitu fitur dalam bahasa Go yang memungkinkan eksekusi paralel (*multithreading*) secara efisien.

Pengujian juga menunjukkan bahwa elemen dengan tingkat (*tier*) yang lebih tinggi cenderung memiliki jumlah pohon solusi yang lebih banyak. Hal ini disebabkan oleh banyaknya elemen penyusun serta kombinasi resep yang memungkinkan untuk membentuk elemen target tersebut.

Adapun jumlah *visited node* yang ditampilkan pada situs merupakan akumulasi dari semua simpul (*node*) yang dieksplorasi selama proses pembentukan pohon, hingga mencapai jumlah sesuai batas input yang ditentukan, atau hingga seluruh kemungkinan pohon valid berhasil dibentuk jika batas input melebihi jumlah solusi yang tersedia.

BAB 5
KESIMPULAN DAN SARAN

5.1. Kesimpulan

Dalam tubes ini, kelompok akuAdalahStima2 berhasil menciptakan algoritma Breadth-First Search (BFS) dan Depth-First Search (DFS) untuk mencari rute pembuatan suatu elemen sesuai dengan nama target elemen dan banyaknya variasi solusi. Seluruhnya dikemas dalam suatu website. Website yang dibentuk menggunakan *framework* ReactJS dan bahasa Go untuk bagian backend. Beberapa percobaan dilakukan pada algoritma BFS dan DFS untuk menghasilkan pohon yang valid dan optimal.

Secara keseluruhan, dapat disimpulkan bahwa penerapan algoritma BFS dan DFS menghasilkan jumlah multiple recipe yang sama. Perbedaan antara kedua algoritma ini dapat dilihat secara lebih intuitif pada visualisasi tree yang tertampil saat proses pencarian selesai. BFS juga menerapkan penggunaan queue sedangkan DFS menggunakan proses iteratif dan memungkinkan *backtracking*. Waktu yang digunakan untuk proses pencarian tidak terlalu lama, hanya membutuhkan beberapa ms. Hal ini selain dipengaruhi oleh algoritma, juga diakibatkan penggunaan multithreading, yaitu goroutine (secara spesifik merupakan fungsi pada bahasa Go yang mengatur multithreading).

5.2. Saran dan Refleksi

Dari Tugas Besar 2 IF2211 Strategi Algoritma ini, kami menyarankan agar pembuatan spesifikasi dibuat lebih jelas dan rinci sehingga tidak banyak revisi dan pengecekan QnA pada proses pengerjaan.

Terdapat beberapa refleksi terhadap kelompok akuAdalahStima2 yang bisa menjadi bahan evaluasi untuk perbaikan kedepannya.

1. Pengoptimalan alokasi waktu dalam pengerjaannya untuk meningkatkan hasil program.
2. Perluasan pengembangan algoritma agar menghasilkan algoritma yang benar serta lebih efisien dan optimal.
3. Tidak mengerjakan tugas mendekati deadline.
4. Lebih bahagia dan ceria dalam pengerjaan tugas.

LAMPIRAN

No	Poin	Ya	Tidak
1	Aplikasi dapat dijalankan.	✓	
2	Aplikasi dapat memperoleh data recipe melalui scraping.	✓	
3	Algoritma Depth First Search dan Breadth First Search dapat menemukan recipe elemen dengan benar.	✓	
4	Aplikasi dapat menampilkan visualisasi recipe elemen yang dicari sesuai dengan spesifikasi.	✓	
5	Aplikasi mengimplementasikan multithreading.	✓	
6	Membuat laporan sesuai dengan spesifikasi.	✓	
7	Membuat bonus video dan diunggah pada Youtube.	✓	
8	Membuat bonus algoritma pencarian Bidirectional.		✓
9	Membuat bonus Live Update.		✓
10	Aplikasi di-containerize dengan Docker.	✓	
11	Aplikasi di-deploy dan dapat diakses melalui internet.		✓

Tautan Repository GitHub

Frontend : https://github.com/AlfianHanifFY/Tubes2_FE_akuAdalahStima2

Backend : https://github.com/AlfianHanifFY/Tubes2_BE_akuAdalahStima2

Tautan Video

Youtube : https://youtu.be/4bLRJbNqU24

DAFTAR PUSTAKA

Munir, R. 2025. “Breadth First Search (BFS) dan Depth First Search (DFS) - Bagian 1”. [*Online*].Tersedia:https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf . [12 Mei 2025].

Munir, R. 2025. “Breadth First Search (BFS) dan Depth First Search (DFS) - Bagian 2”. [*Online*].Tersedia:https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian2.pdf . [12 Mei 2025].

(n.d.). React. Retrieved May 12, 2025, from https://react.dev/

Barney, N. (n.d.). *What Is the Go Programming Language (Golang)? | Definition from TechTarget*. TechTarget. Retrieved May 12, 2025, from https://www.techtarget.com/searchitoperations/definition/Go-programming-language

Documentation. (n.d.). The Go Programming Language. Retrieved May 12, 2025, from https://go.dev/doc/

Jain, S. (2025, March 29). *Depth First Search or DFS for a Graph*. GeeksforGeeks. Retrieved May 12, 2025, from https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

Jain, S. (2025, April 21). *Breadth First Search or BFS for a Graph*. GeeksforGeeks. Retrieved May 12, 2025, from https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/