

**LAPORAN TUGAS BESAR 2**  
**IF2211 STRATEGI ALGORITMA**  
**PEMANFAATAN ALGORITMA BFS DAN DFS DALAM PENCARIAN RECIPE PADA**  
**PERMAINAN LITTLE ALCHEMY 2**



Disusun Oleh:  
Kelompok 34 - STEIcu

Wardatul Khoiroh	13523001
Najwa Kahani Fatima	13523043
Noumisyifa Nareswari	13523058

**PROGRAM STUDI TEKNIK INFORMATIKA**  
**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**  
**INSTITUT TEKNOLOGI BANDUNG**  
**JL. GANESA 10, BANDUNG, 40132**

**2025**

## DAFTAR ISI

<b>DESKRIPSI MASALAH.....</b>	<b>3</b>
<b>LANDASAN TEORI.....</b>	<b>5</b>
<b>2.1. Penjelajahan Graf.....</b>	<b>5</b>
2.2. Breadth First Search (BFS).....	5
2.3. Depth First Search (DFS).....	6
2.4. Bidirectional Search.....	7
2.5. Pengembangan Website.....	7
2.5.1 Pengembangan Web Front-end.....	7
2.5.2 Pengembangan Web Back-end.....	8
2.6. Kakas dan Bahasa Pemrograman.....	8
2.6.1 Go Language.....	8
2.6.2 Multithreading.....	8
2.6.3 Next.js dan JavaScript.....	8
2.6.4 Data Scraping.....	9
2.6.5 Docker.....	9
<b>ANALISIS PEMECAHAN MASALAH.....</b>	<b>10</b>
3.1 Langkah-langkah Pemecahan Masalah.....	10
3.2 Proses Pemetaan Masalah.....	10
3.2.1 Pemetaan Masalah dengan BFS.....	11
3.2.2 Pemetaan Masalah dengan DFS.....	12
3.2.3 Pemetaan Masalah dengan Bidirectional.....	12
3.3 Fitur Fungsional dan Arsitektur Aplikasi Web.....	14
3.4 Contoh Ilustrasi Kasus.....	14
3.4.1 Ilustasi Kasus BFS.....	15
3.4.2 Ilustasi Kasus DFS.....	15
3.4.3 Ilustasi Kasus Bidirectional.....	15
<b>IMPLEMENTASI DAN PENGUJIAN.....</b>	<b>16</b>
4.1 Implementasi Back-end: Struktur Data Umum.....	16
4.2 Implementasi Back-end: Algoritma BFS.....	19
4.3 Implementasi Back-end: Algoritma DFS.....	27
4.4 Implementasi Back-end: Algoritma Bidirectional.....	38
4.5 Implementasi Back-end: Server dan Data Scraping.....	58
4.6 Tata Cara Penggunaan Program.....	59
4.7 Hasil Pengujian dan Analisis.....	61

<b>KESIMPULAN DAN SARAN.....</b>	<b>62</b>
5.1    Kesimpulan.....	62
5.2    Saran.....	62
<b>LAMPIRAN.....</b>	<b>63</b>
<b>DAFTAR PUSTAKA.....</b>	<b>65</b>

# BAB I

## DESKRIPSI MASALAH



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 pertama Strategi Algoritma ini, mahasiswa diminta untuk menyelesaikan permainan Little Alchemy 2 ini dengan menggunakan strategi Depth First Search dan Breadth First Search. 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 dapat melakukan *combine* antara 2 elemen untuk menghasilkan elemen baru. Elemen turunan yang telah didapatkan dapat digunakan kembali oleh pemain untuk membentuk elemen lainnya.

## BAB II

### LANDASAN TEORI

#### 2.1. Penjelajahan Graf

Algoritma penjelajahan (traversal) graf merupakan proses mengunjungi simpul-simpul di dalam graf dengan cara yang sistematik. Traversal graf artinya melakukan pencarian solusi persoalan yang direpresentasikan dengan graf. Algoritma pencarian solusi berbasis graf terbagi menjadi dua berdasarkan pengetahuan sistem terhadap graf, yakni:

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

Pada pencarian tanpa informasi, sistem tidak mengetahui informasi apapun mengenai graf yang akan ditelusuri. Tidak ada informasi tambahan yang disediakan. Contoh algoritma yang termasuk jenis ini adalah Depth First Search (DFS), Breadth First Search (BFS), Depth Limited Search (DLS), Iterative Deepening Search (IDS), dan Uniform Cost Search (UCS).

2. Dengan informasi (*informed search*)

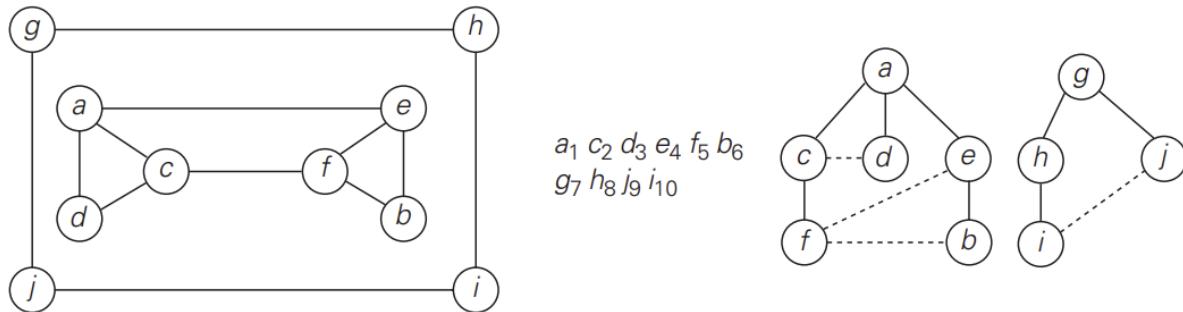
Pada pencarian dengan informasi, sistem menggunakan pencarian berbasis heuristik dengan mengetahui *non-goal state* yang lebih menjanjikan daripada yang lain. Contoh dari algoritma penjelajahan graf yang termasuk jenis ini adalah Best First Search dan A\*.

Dalam proses pencarian solusi, terdapat dua pendekatan, yakni graf statis dan graf dinamis. Graf statis merupakan graf yang sudah terbentuk sebelum proses pencarian dilakukan yaitu graf dengan representasi struktur data. Graf dinamis merupakan graf yang terbentuk saat proses pencarian dilakukan, sehingga graf dibangun selama pencarian solusi.

#### 2.2. Breadth First Search (BFS)

Algoritma Breadth First Search (BFS) berlangsung secara konsentris dengan mengunjungi terlebih dahulu semua simpul yang berdekatan dengan simpul awal, kemudian semua simpul yang belum dikunjungi yang berjarak dua sisi darinya, dan seterusnya, hingga semua simpul dalam komponen terhubung yang sama dengan simpul awal dikunjungi. Jika masih ada simpul yang belum dikunjungi, algoritma harus dimulai ulang pada simpul sembarang dari komponen terhubung lain dari grafik. Akan lebih mudah menggunakan antrian (*queue*) untuk melacak operasi BFS. Antrean diinisialisasi dengan simpul awal penelusuran, yang ditandai sebagai telah dikunjungi. Pada setiap iterasi, algoritma mengidentifikasi semua simpul yang belum dikunjungi

yang berdekatan dengan simpul depan, menandainya sebagai telah dikunjungi, dan menambahkannya ke antrian; setelah itu, simpul depan dihapus dari antrian.

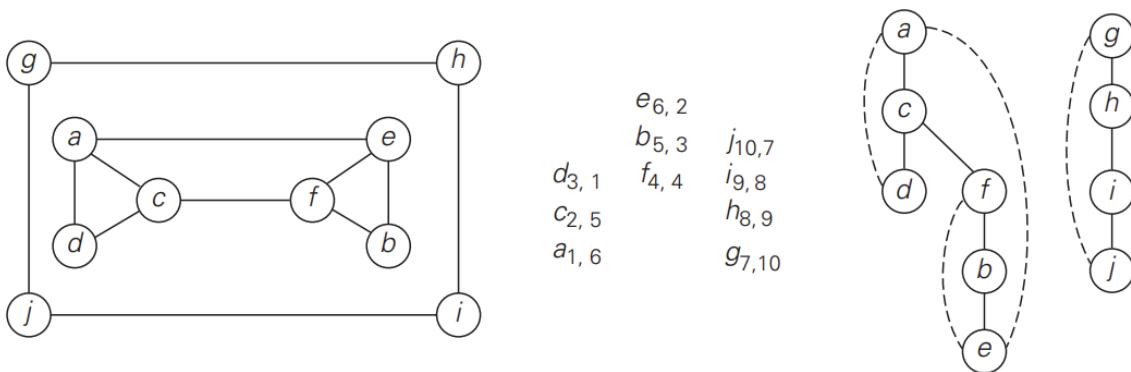


Gambar 3. Ilustrasi BFS

(sumber: Introduction to The Design and Analysis of Algorithms, Anany Levitin)

### 2.3. Depth First Search (DFS)

Algoritma Depth First Search (DFS) memulai penelusuran graf pada simpul sembarang dengan menandainya sebagai telah dikunjungi. Pada setiap iterasi, algoritma berlanjut ke simpul berdekatan yang belum dikunjungi dari simpul saat ini. Proses ini berlanjut hingga jalan buntu—tidak ada simpul yang berdekatan yang belum dikunjungi—ditemukan. Di jalan buntu, algoritma kembali ke satu sisi ke simpul asalnya (*backtrack*) dan mencoba untuk terus mengunjungi simpul lain yang belum dikunjungi. Algoritma DFS berhenti setelah kembali ke simpul awal, dengan titik yang terakhir menjadi jalan buntu, atau ketika suatu solusi yang diharapkan sudah ditemukan. Saat itu, semua simpul dalam komponen terhubung ke simpul awal.

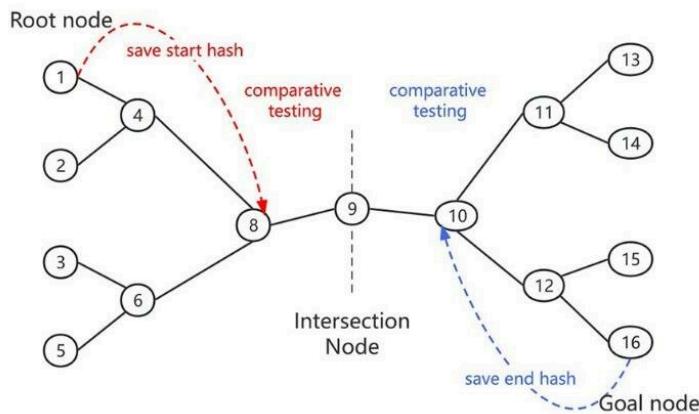


Gambar 4. Ilustrasi DFS

(sumber: Introduction to The Design and Analysis of Algorithms, Anany Levitin)

## 2.4. Bidirectional Search

Bidirectional search merupakan salah satu algoritma pencarian dua arah yang menggunakan konsep *back to front* dan *front to back*. Pencarian dilakukan dari simpul awal graf sampai simpul terakhir dan pencarian dari simpul akhir graf ke simpul awal. Pencarian akan berakhir ketika kedua pencarian bertemu di tengah graf.



Gambar 5. Ilustrasi Bidirectional Search

(sumber: Combinatorial Optimization Design of Search Tree Model Based on Hash Storage, Yun Liu, Jiajun Li, Jingjing Chen)

## 2.5. Pengembangan Website

Pengembangan website secara umum terdiri atas pengembangan antarmuka (*front-end*) dan *back-end*. Dengan website, pengguna dapat dengan mudah menggunakan layanan yang diberikan oleh program ini.

### 2.5.1 Pengembangan Web *Front-end*

*Front-end* merupakan istilah yang digunakan untuk menjelaskan tampilan antarmuka komputer yang memungkinkan interaksi dengan pengguna. Bagian ini berperan penting dalam menyajikan aspek visual dari suatu sistem, termasuk desain layout, pemilihan warna, dan tipografi. Umumnya, *front-end* dikembangkan sebagai pelengkap dari keseluruhan sistem. Dalam prosesnya, pengembangan *front-end* dimulai dengan melakukan desain *user interface* atau antarmuka pengguna, yang biasanya dilakukan melalui aplikasi Figma. Implementasi *front-end* biasanya dilakukan dengan bahasa HTML, CSS, dan JavaScript.

### **2.5.2 Pengembangan Web *Back-end***

Pada suatu sistem atau layanan, saat aplikasi dijalankan data bisa ditambahkan, dimodifikasi, atau dihapus. *Back-end* berperan dalam menangani berbagai proses yang tidak melibatkan interaksi langsung dengan pengguna, seperti pengelolaan server dan *database*. Seluruh logika data dan program berjalan di bagian ini. Keberadaan *back-end* sangat penting dalam proses pengembangan sistem serta pengelolaan data di dalamnya.

## **2.6. Kakas dan Bahasa Pemrograman**

Berbagai bahasa pemrograman dan kakas yang ada dapat digunakan secara bersamaan untuk menciptakan program yang sesuai dengan kebutuhan. Pada bagian ini, akan dijelaskan beberapa kakas dan bahasa pemrograman yang digunakan.

### **2.6.1 Go Language**

Go merupakan bahasa pemrograman yang efisien, sederhana, dan mudah dipahami. Dengan fitur konkurensinya, pengembang dapat menulis program yang optimal untuk mesin *multicore* dan sistem berbasis jaringan. Sistem tipenya yang unik mendukung pengembangan program secara modular dan fleksibel. Meskipun Go menghasilkan kode mesin dengan cepat, ia tetap menyediakan fitur seperti pengumpulan sampah otomatis dan refleksi saat program dijalankan. Bahasa ini menggabungkan kecepatan dan keunggulan tipe statis seperti bahasa yang dikompilasi, namun tetap memberikan pengalaman seperti bahasa bertipe dinamis yang diinterpretasikan.

### **2.6.2 Multithreading**

Multithreading secara umum adalah sebuah fitur yang memungkinkan untuk sebuah program menjalankan lebih dari satu proses dalam waktu yang bersamaan. Dalam konteks pencarian dalam struktur data *tree*, multithreading digunakan untuk melakukan pencarian pada lebih dari satu cabang secara bersamaan yang tentunya akan meningkatkan efektifitas dari program tersebut. Menyinggung kembali bahasa Go, bahasa ini sudah memiliki fitur *multithreading built-in feature* yang direpresentasikan di Goroutines. Dalam Golang, Go routines memungkinkan pengembang untuk mencapai eksekusi serentak dan secara efektif memanfaatkan beberapa utas untuk pemrosesan yang efisien. Dengan Goroutines, pengembang dapat mengeksekusi beberapa fungsi atau blok kode secara serentak, memanfaatkan sepenuhnya prosesor multicore modern.

### **2.6.3 Next.js dan JavaScript**

Next.js merupakan framework berbasis React yang digunakan dalam pengembangan aplikasi web *full-stack*. Dengan menggunakan komponen React untuk merancang

tampilan antarmuka, Next.js menyediakan berbagai fitur tambahan dan pengoptimalan yang memudahkan proses pengembangan. Framework ini secara otomatis menangani konfigurasi alat-alat teknis seperti bundler dan compiler, sehingga pengembang dapat lebih fokus pada pembuatan produk dan mempercepat waktu rilis. Next.js cocok digunakan baik oleh pengembang tunggal maupun tim besar dalam menciptakan aplikasi React yang responsif, dinamis, dan berkinerja tinggi.

JavaScript merupakan bahasa pemrograman dinamis yang mendukung berbagai gaya pemrograman, lengkap dengan tipe data, operator, objek standar bawaan, serta metode. Sintaks JavaScript terinspirasi dari bahasa Java dan C, sehingga banyak struktur dari kedua bahasa tersebut juga digunakan dalam JavaScript. Bahasa ini mendukung pemrograman berorientasi objek melalui penggunaan prototipe dan kelas, serta mendukung paradigma pemrograman fungsional karena fungsi dalam JavaScript diperlakukan sebagai objek tingkat pertama yang dapat dibuat, digunakan, dan dipindahkan layaknya objek biasa.

#### 2.6.4 Data Scraping

*Web scraping (data scraping)* merupakan teknik otomatis yang digunakan untuk mengumpulkan data dalam jumlah besar dari situs web. Umumnya, data yang diperoleh masih dalam bentuk tidak terstruktur (HTML), sehingga perlu dikonversi menjadi data terstruktur agar dapat disimpan di spreadsheet atau basis data dan digunakan dalam berbagai keperluan. Pengambilan data ini bisa dilakukan melalui berbagai metode, seperti memanfaatkan layanan online, menggunakan API khusus, atau dengan membangun skrip scraping sendiri dari awal.

#### 2.6.5 Docker

Docker merupakan sebuah platform yang memudahkan proses pembangunan, pengujian, dan penyebaran aplikasi secara efisien. Dengan menggunakan kontainer, Docker mengemas seluruh komponen yang diperlukan aplikasi seperti pustaka, alat sistem, kode, dan runtime ke dalam satu paket standar. Melalui Docker, pengembang dapat dengan cepat menjalankan dan mengatur aplikasi di berbagai lingkungan tanpa khawatir tentang ketidakcocokan sistem.

## BAB III

### ANALISIS PEMECAHAN MASALAH

#### 3.1 Langkah-langkah Pemecahan Masalah

Permasalahan yang akan diselesaikan pada Tugas Besar ini adalah pencarian resep dari suatu elemen yang ada pada Little Alchemy 2. Terdapat tiga algoritma yang dapat dipilih untuk mencari resep tersebut, yakni 1) Breadth First Search, 2) Depth First Search, dan 3) Bidirectional. Untuk masing-masing algoritma pencarian, terdapat pula opsi pencarian satu resep (*single recipe*) dan banyak resep (*multiple recipe*) yang mampu menerima permintaan jumlah resep yang dicari oleh pengguna.

Setiap pertama kali dijalankan, program akan melakukan data scraping dari laman informasi resep Little Alchemy 2. Saat data scraping berjalan, dilakukan juga penyaringan resep yang dapat masuk dalam sistem program, yakni resep dengan elemen-elemen penyusunnya memiliki *tier* di bawah elemen target yang dicari. Hasil scraping akan disimpan dalam sebuah file berformat .json untuk nantinya dapat digunakan sebagai sumber data pencarian resep.

Program akan menerima input dari pengguna berupa pilihan algoritma, metode, dan jumlah resep yang ingin dicari (jika memilih metode *multiple recipe*). Input parameter dari pengguna melalui *front-end* dikirimkan ke *back-end* dengan Untuk metode *multiple recipe*, pencarian akan dilakukan dengan *multithreading*, baik untuk DFS maupun BFS. Ketika menekan tombol “Search”, sistem akan melakukan pencarian dan mengembalikan hasil resep dalam bentuk *tree* ke *front-end* beserta informasi tambahan seperti durasi pencarian, banyaknya simpul yang dikunjungi selama pencarian, dan banyaknya resep yang ditemukan (jika memilih metode *multiple recipe*). Lalu, *front-end* akan menampilkan pohon penyelesaian pada layar.

Program ini diimplementasikan dalam bentuk website dengan bahasa Go untuk *back-end* dan algoritma pencarian resep, serta JavaScript dengan *framework* Next.js untuk *front-end*.

#### 3.2 Proses Pemetaan Masalah

Proses pemetaan masalah untuk menemukan resep target elemen berbeda-beda tergantung pada algoritma yang digunakan. Secara umum, elemen dasar yang dapat menjadi simpul daun dari pohon solusi adalah “Air”, “Earth”, “Fire”, dan “Water” dengan simpul akar pohon adalah elemen target yang ingin diketahui resepnya. Sehingga, kondisi ini sering kali menjadi parameter penentuan solusi. Setiap simpul menyimpan informasi nama elemen dan kombinasi dari resep, dengan setiap resep terdiri atas dua simpul lainnya.

Pemetaan masalah dengan algoritma BFS dan DFS termasuk dalam kategori graf dinamis karena graf (dalam kasus ini pohon) terbentuk selama pencarian berlangsung. Sedangkan pada Bidirectional, graf telah dibuat sebelum dilakukan pencarian, sehingga termasuk ke dalam kategori graf statis.

Pada pencarian solusi ini, diterapkan juga beberapa heuristik, seperti pembatasan *tier* elemen resep pembentuk suatu elemen target yang harus berada di level (*tier*) lebih bawah dan memoisasi, yakni teknik pengoptimalan dalam pemrograman yang pada kasus ini menyimpan pohon solusi suatu elemen yang telah dikunjungi hingga ke elemen dasar. Berikut adalah proses untuk masing-masing algoritma.

### 3.2.1 Pemetaan Masalah dengan BFS

Sebelum pencarian dimulai, perlu diinisialisasi map untuk memoisasi subpohon solusi dari elemen yang dikunjungi dan map untuk melacak simpul-simpul yang telah dikunjungi. Memoisasi dilakukan untuk mempercepat pencarian dan pelacakan dilakukan untuk menghindari *loop* atau *cyclic* dalam konstruksi pohon solusi. Algoritma ini memanfaatkan struktur data *queue* untuk menyimpan simpul selanjutnya yang harus dieksplorasi.

Untuk metode *single recipe*, di awal pencarian, elemen target menjadi simpul akar, lalu elemen pada kombinasi-kombinasi resep dari target di-*enqueue* ke dalam *queue* pengecekan simpul. Pada setiap iterasi, akan dilakukan *dequeue* elemen pertama untuk dicek. Jika simpul untuk elemen tersebut belum dibuat sebelumnya, maka simpul baru akan dibuat dan disimpan. Jika elemen yang dicek adalah elemen dasar, maka iterasi otomatis dilanjutkan ke elemen selanjutnya tanpa proses lebih lanjut. Jika bukan elemen dasar, maka setiap elemen dari kombinasi resep elemen tersebut, akan dimasukkan ke dalam antrian *queue*. Iterasi akan terus dilakukan hingga antrian simpul yang harus dicek habis pada *queue*. Setelah itu, dilakukan konstruksi pohon solusi setelah seluruh simpul terbuat.

Untuk metode *multiple recipe*, pencarian BFS dilakukan dengan *multithreading*. Setiap resep (yang terdiri dari dua bahan) diproses secara paralel menggunakan goroutine. Masing-masing goroutine memperluas bahan-bahan resep menjadi node-node pohon yang juga bekerja secara paralel untuk mengeksplorasi sub-resep secara rekursif. Hasil node yang membentuk jalur pembuatan target dikumpulkan melalui channel hingga mencapai jumlah maksimum jalur. Setelah seluruh proses selesai, program mengkonversi node-node hasil pencarian menjadi struktur pohon dan menghitung jumlah elemen pada tiap jalur sebelum mengembalikannya sebagai hasil akhir.

### 3.2.2 Pemetaan Masalah dengan DFS

Proses pencarian resep dimulai dari elemen target menjadi *root* dari pohon solusi. Kemudian secara rekursif dilakukan penelusuran kombinasi bahan yang tersedia untuk membentuk elemen tersebut. Sebelum pencarian dimulai, beberapa struktur data diinisialisasi, seperti memo untuk menyimpan hasil pencarian sebelumnya (memoisasi) dan map untuk mendeteksi siklus serta menghindari *infinite loop*.

Untuk *single recipe*, fungsi dfsOne dipanggil secara rekursif untuk setiap elemen, dimulai dari elemen kombinasi pertama dari target. Jika elemen tersebut merupakan bahan dasar, maka akan langsung dikembalikan sebagai simpul daun. Jika elemen memiliki resep, maka fungsi akan memanggil dirinya sendiri (rekursif) untuk masing-masing elemen penyusunnya. Jika kedua bahan berhasil ditemukan tanpa menimbulkan siklus, maka simpul hasil gabungan akan dibentuk dan ditambahkan ke hasil akhir. Proses akan berhenti setelah satu pohon resep valid berhasil dibentuk. Selain membangun pohon solusi, program juga akan mencatat jumlah simpul unik yang dikunjungi selama proses pencarian.

Metode *multiple recipe* DFS dioptimasi menggunakan *multithreading*. Proses dimulai dengan menelusuri semua kombinasi resep yang tersedia untuk elemen target. Untuk setiap kombinasi, dilakukan eksplorasi dua cabang (untuk masing-masing elemen dalam resep) secara paralel menggunakan Goroutine. Algoritma ini menerapkan DFS rekursif yang dilengkapi dengan pendekripsi siklus dan batasan kedalaman maksimum agar mencegah rekursi yang tidak terbatas atau *loop*. Pada tingkat kedalaman rendah, DFS dilakukan secara paralel untuk mempercepat pencarian, sedangkan di tingkat yang lebih dalam, pencarian dilakukan secara linear dengan pembatasan jumlah kombinasi yang diperiksa. Untuk menghindari redundansi dan pemborosan sumber daya, hanya sebagian kecil kombinasi yang dijelajahi berdasarkan heuristik. Hasil dari setiap pencarian DFS dikombinasikan membentuk pohon solusi yang lengkap dan unik.

### 3.2.3 Pemetaan Masalah dengan Bidirectional

Sebelum pencarian dimulai, algoritma ini memerlukan struktur *tree* yang merepresentasikan semua kemungkinan resep dari elemen target hingga elemen dasar. Pohon ini dibangun menggunakan pendekatan BFS. Setelah pohon awal terbentuk, pencarian dua arah dapat dimulai.

Algoritma ini menginisialisasi dua set struktur data untuk melakukan pencarian secara bersamaan dari dua arah:

1. Pencarian Maju (Forward Search): Dimulai dari simpul akar (elemen target). Menggunakan sebuah queue (`q_f`) untuk menyimpan simpul yang akan dieksplorasi dan sebuah map (`visited_f`) untuk melacak simpul yang telah dikunjungi dari arah depan serta menyimpan simpul pendahulunya (parent dalam jalur pencarian) untuk rekonstruksi jalur. Juga menggunakan map `forwardDepth` untuk menyimpan kedalaman simpul dari akar.
2. Pencarian Mundur (Backward Search): Dimulai secara simultan dari semua simpul daun yang merupakan elemen dasar. Menggunakan queue kedua (`q_b`) dan map kedua (`visited_b`) untuk melacak simpul yang dikunjungi dari arah belakang dan pendahulunya (child dalam jalur pencarian). Juga menggunakan map `backwardDepth` untuk menyimpan kedalaman simpul dari daun dasar terdekat.

Pencarian dilakukan secara bergantian atau paralel tiap satu loop dengan langkah berikut:

1. Langkah Maju: Ambil simpul, yaitu `curr_f_instance` dari `q_f`. Periksa apakah simpul ini sudah dikunjungi oleh pencarian mundur dengan mengecek keberadaannya di `backwardDepth`. Jika ya, simpul pertemuan telah ditemukan, dan pencarian berhenti. Jika tidak, eksplorasi anak-anak dari `curr_f_instance`. Untuk setiap anak yang belum dikunjungi oleh pencarian maju, tambahkan ke `q_f`, catat `curr_f_instance` sebagai pendahulunya di `visited_f`, dan perbarui kedalamannya di `forwardDepth`. Setelah menambahkan anak, periksa lagi apakah anak tersebut sudah dikunjungi oleh pencarian mundur, jika ya, simpul pertemuan ditemukan. Simpul yang ditandai sebagai bagian dari siklus akan dilewati.
2. Langkah Mundur: Ambil simpul, yaitu `curr_b_instance` dari `q_b`. Periksa apakah simpul ini sudah dikunjungi oleh pencarian maju dengan mengecek keberadaannya di `forwardDepth`. Jika ya, simpul pertemuan ditemukan, dan pencarian berhenti. Jika tidak, eksplorasi induk dari `curr_b_instance`. Jika induk ada, bukan cycle node, dan belum dikunjungi oleh pencarian mundur, tambahkan ke `q_b`, catat `curr_b_instance` sebagai pendahulunya (dari arah daun) di `visited_b`, dan perbarui kedalamannya di `backwardDepth`. Setelah menambahkan induk, periksa lagi apakah induk tersebut sudah dikunjungi oleh pencarian maju; jika ya, simpul pertemuan ditemukan.

Iterasi berlanjut hingga salah satu queue kosong atau simpul pertemuan ditemukan. Setelah simpul pertemuan (meetingNode) ditemukan, dilakukan konstruksi pohon solusi:

1. Jalur dari akar ke `meetingNode` direkonstruksi dengan menelusuri balik peta `visited_f`.
2. Jalur dari daun dasar ke `meetingNode` direkonstruksi dengan menelusuri balik peta `visited_b`.

3. Kedua jalur ini digabungkan untuk membentuk jalur lengkap terpendek dari akar ke salah satu daun dasar.
4. Sebuah pohon solusi baru dibangun berdasarkan jalur yang ditemukan ini. Proses ini melibatkan pembuatan salinan simpul-simpul di jalur dan menambahkan kombinasi resep yang relevan untuk merekonstruksi langkah-langkah crafting spesifik di sepanjang jalur tersebut.

### 3.3 Fitur Fungsional dan Arsitektur Aplikasi Web

Website Little Alchemy 2 Recipe Finder - STEIcu dibangun menggunakan Next.js sebagai framework frontend dengan JavaScript (JSX). Halaman utama website menyediakan antarmuka bagi pengguna untuk memilih algoritma pencarian yang akan digunakan untuk menemukan resep pembentukan elemen dalam permainan Little Alchemy 2. Pada halaman ini, pengguna perlu mengetikkan nama elemen yang ingin dicari. Nama elemen ini akan dikirimkan ke server backend melalui permintaan HTTP. Bagian backend dikembangkan dengan bahasa pemrograman Go menggunakan framework Gin untuk membangun web server. Server mendefinisikan endpoint GET pada root path (/) yang menerima parameter query berupa target elemen yang ingin dicari. Jika parameter ini tidak disertakan, server akan mengembalikan respon HTTP 400 (Bad Request).

Setelah menerima input yang valid, backend akan menjalankan proses pencarian resep menggunakan algoritma pencarian BFS, DFS, atau Bidirectional. Backend memanfaatkan web scraping secara paralel dengan menjalankan beberapa goroutine untuk mengambil dan memproses kombinasi elemen yang mungkin, mengelola elemen yang telah dan belum dikunjungi dalam struktur antrian. Setelah rute pembuatan elemen ditemukan (atau waktu pencarian habis), server akan mengonversi path kombinasi elemen menjadi tree yang dapat dipahami pengguna, serta menghitung waktu eksekusi, jumlah kombinasi node yang dikunjungi, dan banyak resep yang dicari/ditemukan. Seluruh hasil ini kemudian dikembalikan ke frontend dalam bentuk objek JSON melalui fungsi c.JSON.

### 3.4 Contoh Ilustrasi Kasus

Pada contoh ini, akan dicari resep dari elemen “Stone”. Berikut adalah data dari hasil scraping yang sesuai dengan pembentukan elemen “Stone”.

```
"Stone": {  
    "Lava": [[ "Earth", "Fire" ]],  
    "Pressure": [[ "Air", "Air" ] ],  
    "Stone": [[ "Earth", "Pressure"], [“Air”, “Lava”]]  
}
```

### 3.4.1 Ilustasi Kasus BFS

Ketika fungsi dijalankan dengan target "Stone", proses pencarian dilakukan menggunakan pendekatan Breadth-First Search (BFS), yang berarti semua node pada level tertentu diproses terlebih dahulu sebelum melanjutkan ke level berikutnya. Pertama, "Stone" dimasukkan ke dalam antrean. Karena "Stone" bukan elemen dasar, kedua resepnya, yaitu ["Earth", "Pressure"] dan ["Air", "Lava"], diperiksa, tetapi hanya resep pertama ([["Earth", "Pressure"]]) yang disimpan untuk digunakan nanti (karena BFS hanya memilih satu kombinasi pertama yang valid). Selanjutnya, "Earth" dan "Pressure" dimasukkan ke antrean. "Earth" langsung dikenali sebagai elemen dasar, sedangkan "Pressure" diproses lebih lanjut dan memiliki satu resep ["Air", "Air"]. Kedua "Air" juga dimasukkan ke antrean. Setelah semua elemen dan resepnya dicatat dalam urutan BFS, algoritma mulai menyusun pohon dari resep pertama yang ditemukan untuk setiap elemen. "Pressure" dibentuk dari dua "Air", lalu "Stone" dibentuk dari "Earth" dan "Pressure". Hasil akhirnya adalah pohon kombinasi Stone → Earth + Pressure → Air + Air, yang diperoleh melalui eksplorasi BFS dengan pencatatan semua node terlebih dahulu sebelum konstruksi dimulai.

### 3.4.2 Ilustasi Kasus DFS

Ketika fungsi pencarian dipanggil dengan target "Stone", algoritma akan melakukan pencarian jalur pembuatan satu-satunya pohon kombinasi berdasarkan pendekatan Depth-First Search (DFS). Pertama, fungsi akan mencoba resep pertama untuk "Stone", yaitu ["Earth", "Pressure"]. Karena "Earth" adalah elemen dasar, maka node untuk "Earth" dibuat. Selanjutnya, DFS menelusuri "Pressure", yang memiliki resep ["Air", "Air"]; karena "Air" adalah elemen dasar, maka dua node "Air" dibuat dan digabung menjadi node "Pressure". Setelah kedua bahan "Earth" dan "Pressure" ditemukan, node "Stone" dibentuk dari kombinasi tersebut, dan pencarian berhenti karena DFS hanya mencari satu jalur sukses pertama. Dengan demikian, jalur yang dihasilkan adalah: Stone → Earth + Pressure → Air + Air. Resep alternatif kedua untuk "Stone" yaitu [Air, "Lava"] tidak dieksplorasi karena DFS telah menemukan jalur yang valid dari percobaan pertama.

### 3.4.3 Ilustasi Kasus Bidirectional

## BAB IV

### IMPLEMENTASI DAN PENGUJIAN

#### 4.1 Implementasi Back-end: Struktur Data Umum

Di bawah ini adalah struktur data dan fungsi atau prosedur yang ada pada tree.go sebagai struktur data umum yang digunakan utamanya untuk pohon solusi.

##### 4.1.1 Struktur Data

Struktur Data	Penjelasan
Node <pre>type Node struct {     element      string     combinations []Recipe }</pre>	Struktur data Node yang dibentuk dengan struct dan berisi element bertipe string serta combinations yang bertipe array of Recipe. Node merepresentasikan elemen.
Recipe <pre>type Recipe struct {     ingredient1 *Node     ingredient2 *Node }</pre>	Struktur data yang dibuat dari struct dan terdiri atas dua <i>pointer</i> menuju Node. Digunakan untuk menunjukkan suatu kombinasi resep yang terdiri dari dua elemen.
Tree <pre>type Tree struct {     root *Node }</pre>	Struktur data yang berisi root dengan tipe <i>pointer</i> menuju Node. Digunakan untuk
Map (visited) <pre>var visited map[string]bool</pre>	Struktur data map dengan key bertipe string untuk menyimpan nama elemen yang telah dikunjungi dan value boolean.

##### 4.1.2 Fungsi dan Prosedur

1. Fungsi isBase(string) → boolean

Fungi ini mengecek apakah suatu elemen merupakan *base* elemen atau tidak dan mengembalikan boolean.

```
func isBase(element string) bool {
    return (element == "Air" || element == "Earth" || element == "Water" || element == "Fire")
```

```

== "Fire" || element == "Water")
}

```

2. Fungsi initTree(string, map[string][][]string) → \*Tree

Fungsi ini dipanggil untuk menginisiasi pembuatan *tree* dengan menerima elemen target dalam string dan recipeData untuk database resep.

```

func InitTree(target string, recipeData map[string][][]string)
*Tree {
    visited = make(map[string]bool)
    root := buildTree(target, target, recipeData, 0)
    return &Tree{root: root}
}

```

3. Fungsi buildTree(string, string, map[string][][]string, int) → \*Node

Fungsi ini membangun pohon resep dari suatu elemen target secara rekursif dengan menelusuri semua kombinasi bahan penyusunnya berdasarkan data resep. Fungsi ini menggunakan pengecekan terhadap kondisi dasar seperti elemen dasar (isBase), deteksi siklus (menghindari mengunjungi kembali elemen yang sudah dikunjungi melalui visited), serta pembatasan agar elemen target tidak diproses ulang setelah iterasi pertama (cntNode != 0). Untuk setiap pasangan resep, fungsi ini memanggil dirinya sendiri untuk membentuk subtree bahan pertama dan kedua, lalu menggabungkannya sebagai satu Recipe dalam node tersebut.

```

var visited map[string]bool

func buildTree(target string, element string, recipeData
map[string][][]string, cntNode int) *Node {
    if isBase(element) || (element == target && cntNode != 0)
    || visited[element] {
        return &Node{element: element}
    }

    visited[element] = true
    node := &Node{element: element}
    recipes := recipeData[element]

    for _, combination := range recipes {
        ing1 := buildTree(target, combination[0],
        recipeData, cntNode+1)
        ing2 := buildTree(target, combination[1],
        recipeData, cntNode+1)

```

```
        recipe := Recipe{
            ingredient1: ing1,
            ingredient2: ing2,
        }
        node.combinations = append(node.combinations,
recipe)
    }
    return node
}
```

4. Prosedur printTree(\*Tree)

Prosedur printTree adalah wrapper yang memanggil printTreeHelper pada akar pohon dari struktur Tree yang diberikan. Fungsi ini digunakan untuk menampilkan representasi visual dari keseluruhan pohon resep mulai dari node target sebagai akar.

```
func printTree(t *Tree) {
    printTreeHelper(t.root, "", true)
}
```

5. Prosedur printTreeHelper(\*Node, string, bool)

Prosedur ini adalah prosedur rekursif yang bertanggung jawab untuk mencetak pohon resep dalam format visual seperti struktur direktori. Prosedur ini mencetak nama elemen dari setiap node dengan indentasi dan garis cabang untuk menunjukkan hubungan hierarki, dan memanggil dirinya kembali untuk setiap bahan dalam kombinasi resep node tersebut.

```
func printTreeHelper(node *Node, prefix string, isLast bool) {
    if node == nil {
        return
    }

    // current node
    fmt.Println(prefix)
    if isLast {
        fmt.Print("└── ")
        prefix += "    "
    } else {
        fmt.Print("├── ")
        prefix += "|   "
    }
    fmt.Println(node.element)
}
```

```

// combination
for i, recipe := range node.combinations {
    printTreeHelper(recipe.ingredient1, prefix, false)
    isLastRecipe := i == len(node.combinations)-1
    printTreeHelper(recipe.ingredient2, prefix,
isLastRecipe)
}
}

```

## 4.2 Implementasi Back-end: Algoritma BFS

### 4.2.1 Struktur Data

Selain menggunakan struktur data umum yang dijelaskan pada 4.1, implementasi algoritma BPS juga menggunakan struktur data di bawah.

Struktur Data	Penjelasan
Map  map[string]*Node map[string][][]string Map[string]bool sync.Map	Struktur data map[string]*Node (misal memoSB) menyimpan hasil node yang sudah dibentuk agar tidak dihitung ulang. Sedangkan mainDataBFS dan pendingNodes digunakan untuk menyimpan data resep untuk item target, berisi semua kombinasi bahan dan menyimpan pasangan bahan yang masih menunggu pembuatan node-nya. Kedua variabel ini bertipe map[string][][]string. Struktur data map[string]bool (visitedBFS) digunakan untuk menandai item yang sudah dikunjungi dalam proses BFS agar tidak diproses dua kali.
List  []string	Struktur data untuk <i>queue</i> berisi antrian BFS untuk memproses setiap item satu per satu.
Channel  chan *Node	Slice pointer ke node yang menyimpan hasil-hasil pencarian pohon.

### 4.2.2 Fungsi dan Prosedur

1. Fungsi searchBFSOne(string) → (\*Tree, int)

Fungsi ini adalah titik awal pencarian menggunakan algoritma BFS untuk menemukan satu cara pembuatan target elemen. Fungsi ini menginisialisasi struktur pendukung seperti memoSB, mainDataBFS, dan visitedBFS, lalu memanggil fungsi bfsOne untuk membangun pohon resep. Hasilnya adalah struktur Tree yang merepresentasikan cara membuat target item, beserta jumlah simpul (node) yang dikunjungi selama pencarian

```
var memoSB map[string]*Node
var mainDataBFS map[string][][]string
var visitedBFS map[string]bool

func searchBFSOne(target string) (*Tree, int) {
    fmt.Println("start bfs single")
    memoSB = make(map[string]*Node)
    mainDataBFS = recipeData.Recipes[target]
    visitedBFS = make(map[string]bool)

    result, cntNode := bfsOne(target)

    return &Tree{root: result}, cntNode
}
```

2. Fungsi bfsOne(string) → (\*Node, int)

Fungsi ini melakukan pencarian BFS untuk membangun satu pohon resep dari item target. Ia mulai dari target, menyimpan semua bahan dalam antrian (queue), dan menelusuri bahan-bahan tersebut secara bertingkat. Semua kombinasi resep disimpan di pendingNodes, dan simpul-simpul dibentuk setelah semua bahan ditemukan. Akhirnya, fungsi mengembalikan node akar dari pohon resep dan jumlah node unik yang dikunjungi selama pencarian.

```
func bfsOne(element string) (*Node, int) {
    cntNode := 0

    pendingNodes := make(map[string][][]string)
    nodeMap := make(map[string]*Node)

    queue := []string{element}
    visitedBFS[element] = true

    for len(queue) > 0 {
        current := queue[0]
        queue = queue[1:]

        for _, ingred := range pendingNodes[current] {
            if _, ok := visitedBFS[ingred]; !ok {
                visitedBFS[ingred] = true
                pendingNodes[ingred] = append(pendingNodes[ingred], element)
                nodeMap[ingred] = &Node{children: []string{current}}
            }
        }
    }
}
```

```

        if _, exists := nodeMap[current]; !exists {
            nodeMap[current] = &Node{element: current}
        }

        if isBase(current) {
            continue
        }

        if recipes, hasRecipe := mainDataBFS[current];
        hasRecipe && len(recipes) > 0 {
            for _, pair := range recipes {
                if len(pair) != 2 {
                    continue
                }
                pendingNodes[current] =
append(pendingNodes[current], pair)

                for _, ingredient := range pair {
                    cntNode++
                    if !visitedBFS[ingredient] {
                        visitedBFS[ingredient] =
true
                        queue = append(queue,
ingredient)
                    }
                }
            }
        }

        for el, allPairs := range pendingNodes {
            firstPair := allPairs[0]
            ing1 := nodeMap[firstPair[0]]
            ing2 := nodeMap[firstPair[1]]
            nodeMap[el].combinations = []Recipe{
            {
                ingredient1: ing1,
                ingredient2: ing2,
            },
            }
            memoSB[el] = nodeMap[el]
        }
        return nodeMap[element], cntNode
    }
}

```

3. Fungsi searchBFSMultiple(string, int) → ([]\*Tree, []int)

Fungsi utama yang memulai pencarian jalur pembuatan sebuah item (target) dari resep yang tersedia. Fungsi ini mengambil semua resep yang terkait dengan target, lalu memanggil bfsAll untuk mencari jalur-jalur pohon kombinasi secara paralel. Setelah mendapatkan node-node akar hasil pencarian, fungsi ini mengonversinya menjadi struktur Tree dan menghitung jumlah elemen pada setiap jalur sebelum mengembalikannya sebagai daftar pohon dan jumlah elemen per jalur.

```
var allFinalTargetTrees []*Node
func searchBFSSmultiple(target string, maxPathsToReturn int)
([]*Tree, []int) {
    fmt.Println("start bfs multiple")
    targetSpecificRecipes := recipeData.Recipes[target]
    rootNodes := bfsAll(target, maxPathsToReturn,
    targetSpecificRecipes)
    var trees []*Tree
    var pathElementCounts []int
    allFinalTargetTrees = make([]*Node, 0)
    for _, rootNode := range rootNodes {
        trees = append(trees, &Tree{root: rootNode})
        pathElementCounts = append(pathElementCounts,
        getPathElementCount(rootNode))
    }
    return trees, pathElementCounts
}
```

#### 4. Fungsi bfsAll(string, int, map[string][][]string) → []\*Node

Fungsi ini membangun semua kemungkinan pohon kombinasi dari target dengan memproses setiap resep secara paralel menggunakan goroutine. Di setiap goroutine, bahan-bahan dari resep diperluas menjadi node menggunakan expandElementParallel. Hasil node kemudian dikirim melalui channel dan dikumpulkan hingga batas maksimum jumlah jalur tercapai, kemudian dikembalikan sebagai daftar node akar dari pohon-pohon hasil pencarian.

```
func bfsAll(targetElement string, maxPathsToReturn int,
currentRecipeMap map[string][][]string) []*Node {
    var collectedTrees []*Node
    var mu sync.Mutex

    targetTopLevelCombs, exists :=
    currentRecipeMap[targetElement]
    if !exists || len(targetTopLevelCombs) == 0 {
        return []*Node{{element: targetElement}}
    }
```

```

concurrencyLimit := 8
sem := make(chan struct{}, concurrencyLimit)
resultChanBufferSize := 100
if maxPathsToReturn > 0 && maxPathsToReturn <
resultChanBufferSize {
    resultChanBufferSize = maxPathsToReturn
}
resultChan := make(chan *Node, resultChanBufferSize)
var wg sync.WaitGroup
ctx, cancel := context.WithCancel(context.Background())
defer cancel()

pathsFound := 0

for _, topRecipePair := range targetTopLevelCombs {
    if len(topRecipePair) != 2 {
        continue
    }

    sem <- struct{}{}
    wg.Add(1)
    go func(recipe []string, currentContext
context.Context) {
        defer func() {
            <-sem
            wg.Done()
        }()
        pathVisited := make(map[string]bool)
        pathVisited[targetElement] = true
        ing1Name := recipe[0]
        ing2Name := recipe[1]

        expandedIng1Nodes :=
expandElementParallel(ing1Name, currentRecipeMap)
        expandedIng2Nodes :=
expandElementParallel(ing2Name, currentRecipeMap)

        delete(pathVisited, targetElement)

        if len(expandedIng1Nodes) == 0 {
expandedIng1Nodes = []*Node{{element: ing1Name}} }
        if len(expandedIng2Nodes) == 0 {
expandedIng2Nodes = []*Node{{element: ing2Name}} }

        for _, nodeIng1 := range expandedIng1Nodes {
            for _, nodeIng2 := range
expandedIng2Nodes {

```

```

        select {
        case <-currentContext.Done():
            return
        default:
            // continue
        }

        rootNode := &Node{
            element: targetElement,
            combinations: []Recipe{
                ingredient1: nodeIng1,
                ingredient2: nodeIng2,
            }},
        }

        select {
        case resultChan <- rootNode:
        case <-currentContext.Done():
            return
        }
    }
}

}(topRecipePair, ctx)
}
go func() {
    wg.Wait()
    close(resultChan)
}()

for tree := range resultChan {
    mu.Lock()
    if maxPathsToReturn <= 0 || pathsFound <
maxPathsToReturn {
        collectedTrees = append(collectedTrees, tree)
        pathsFound++
        if maxPathsToReturn > 0 && pathsFound >=
maxPathsToReturn {
            cancel()
        }
    }
    mu.Unlock()
}
return collectedTrees
}

```

## 5. Fungsi expandElementParallel(string, map[string][][]string) → []\*Node

Fungsi ini bertugas memperluas sebuah elemen menjadi pohon-pohon kecil berdasarkan resep-resep yang membentuknya, dilakukan secara paralel untuk efisiensi. Dengan memanfaatkan channel pekerjaan dan sync.Map sebagai cache, fungsi ini memproses elemen dan sub-elemennya hingga membentuk node-node yang merepresentasikan kombinasi bahan untuk membentuk elemen tersebut, dan mengembalikan hasilnya dalam bentuk slice dari node.

```
func expandElementParallel(
    elementName string,
    currentRecipeMap map[string][][]string,
) []*Node {
    var memo sync.Map
    var wg sync.WaitGroup
    jobChan := make(chan string, 100)

    worker := func() {
        for elem := range jobChan {
            // fmt.Println(elem)
            if _, exists := memo.Load(elem); exists {
                wg.Done()
                continue
            }

            recipes, ok := currentRecipeMap[elem]
            if !ok || len(recipes) == 0 {
                memo.Store(elem, []*Node{{element:
elem}})
                wg.Done()
                continue
            }

            var nodes []*Node
            for _, recipe := range recipes {
                // fmt.Println("here")
                if len(recipe) != 2 {
                    continue
                }
                ing1 := recipe[0]
                ing2 := recipe[1]

                // Add dependencies
                if _, ok := memo.Load(ing1); !ok {
                    wg.Add(1)
                    jobChan <- ing1
                } else {
                    nodes = append(nodes, &Node{element: elem, ingredient1: ing1, ingredient2: ing2})
                }
            }
            wg.Done()
        }
    }

    go worker()
    for i := 0; i < 10; i++ {
        jobChan <- elementName
    }
    wg.Wait()
}
```

```

        }
        if _, ok := memo.Load(ing2); !ok {
            wg.Add(1)
            jobChan <- ing2
        }

        var ing1Nodes, ing2Nodes []*Node
        for {
            if val, ok := memo.Load(ing1); ok
{
                ing1Nodes = val.([]*Node)
                break
            }
            time.Sleep(time.Millisecond)
        }
        for {
            if val, ok := memo.Load(ing2); ok
{
                ing2Nodes = val.([]*Node)
                break
            }
            time.Sleep(time.Millisecond)
        }

        for _, n1 := range ing1Nodes {
            for _, n2 := range ing2Nodes {
                nodes = append(nodes, &Node{
                    element: elem,
                    combinations:
[]Recipe{{

                        ingredient1: n1,
                        ingredient2: n2,
                    }},
                })
            }
        }
        memo.Store(elem, nodes)
        wg.Done()
    }
}
for i := 0; i < 8; i++ {
    go worker()
}

wg.Add(1)
jobChan <- elementName

```

```

wg.Wait()
close(jobChan)

if val, ok := memo.Load(elementName); ok {
    return val.([]*Node)
}
return []*Node{}
}

```

## 4.3 Implementasi Back-end: Algoritma DFS

### 4.3.1 Struktur Data

Selain menggunakan struktur data umum yang dijelaskan pada 4.1, implementasi algoritma DFS juga menggunakan struktur data di bawah.

Struktur Data	Penjelasan
Map map[string][][]string map[string]*Node map[string]bool	<p>Map map[string][][]string (contoh recipeData.Recipes) menyimpan semua data resep yang tersedia. Key-nya adalah nama item (string), dan value-nya adalah daftar kombinasi resep untuk item tersebut, di mana setiap kombinasi terdiri dari dua bahan ([][]string menandakan banyak kombinasi, masing-masing berisi 2 string).</p> <p>Struktur map[*Node] (contoh memoSD) digunakan untuk memoization pada DFS single. Tujuannya adalah untuk menghindari perhitungan ulang terhadap node yang sudah pernah dibangun sebelumnya.</p> <p>Struktur map boolean digunakan untuk berbagai tujuan</p> <ul style="list-style-type: none"> <li>- visitedDFS: menandai node yang telah dikunjungi dalam DFS.</li> <li>- currentPath: mendeteksi siklus saat DFS sedang berlangsung (mendeteksi loop).</li> <li>- visitedNodes: mencatat semua</li> </ul>

	node yang telah dilewati untuk keperluan perhitungan.
Channel  chan *Node	Channel ini digunakan dalam DFS <i>multiple</i> untuk mengumpulkan hasil pohon dari beberapa goroutine yang berjalan paralel. Channel memungkinkan komunikasi sinkron atau asinkron antar goroutine.
Sync  sync.Mutex sync.WaitGroup	Tipe sync.WaitGroup digunakan untuk menunggu semua goroutine selesai sebelum melanjutkan.  Tipe sync.Mutex digunakan untuk mengatur akses bersama ke map seenStructures agar tidak terjadi kondisi balapan (race condition).

#### 4.3.1 Fungsi dan Prosedur

1. Fungsi searchDFSOne(string) → (\*Tree, int)

Fungsi searchDFSOne merupakan titik awal untuk pencarian satu pohon resep menggunakan pendekatan DFS). Fungsi ini menginisialisasi struktur global seperti memoSD untuk memoization, visitedDFS dan currentPath untuk pelacakan siklus, serta visitedNodes untuk mencatat semua node yang dikunjungi. Kemudian memanggil fungsi dfsOne secara rekursif untuk membangun pohon dari elemen target, dan mengembalikan hasil pohon serta jumlah node yang dikunjungi jika jalur resep valid ditemukan.

```
var memoSD map[string]*Node
var mainData map[string][][]string
var visitedDFS map[string]bool
var currentPath map[string]bool

func searchDFSOne(target string) (*Tree, int) {
    fmt.Println("start dfs single")
    memoSD = make(map[string]*Node)
    mainData = recipeData.Recipes[target]
    visitedDFS = make(map[string]bool)
    currentPath = make(map[string]bool)
    visitedNodes := make(map[string]bool)

    result, found := dfsOne(target, visitedNodes)
```

```

    visitedNodeCount := len(visitedNodes)
    fmt.Println("DFS done")
    if found {
        return &Tree{root: result}, visitedNodeCount
    }
    return nil, 0
}

```

2. Fungsi dfsOne(string, map[string]bool) → (\*Node, bool)

Fungsi dfsOne menjalankan proses DFS rekursif untuk menemukan satu jalur resep valid dari sebuah elemen ke bahan-bahannya. Fungsi ini menggunakan memoisasi (memoSD) untuk efisiensi, serta deteksi siklus dengan currentPath agar tidak terjadi loop tak berhingga. Jika elemen merupakan bahan dasar (isBase), node dikembalikan langsung. Untuk elemen non-dasar, fungsi mencoba setiap kombinasi resep yang tersedia dan membentuk Recipe hanya jika kedua bahan dapat ditemukan secara valid, lalu langsung mengembalikannya begitu kombinasi yang valid ditemukan.

```

func dfsOne(element string, visitedNodes map[string]bool)
(*Node, bool) {
    visitedNodes[element] = true
    visitedDFS[element] = true
    if _, inPath := currentPath[element]; inPath {
        return nil, false
    }
    currentPath[element] = true
    defer delete(currentPath, element)
    if isBase(element) {
        return &Node{element: element}, true
    }
    if res, ok := memoSD[element]; ok {
        return res, true
    }
    res := &Node{element: element, combinations: []Recipe{}}
    memoSD[element] = res
    if recipes, ok := mainData[element]; ok {
        for _, ingredients := range recipes {
            left, leftValid := dfsOne(ingredients[0],
visitedNodes)
            if !leftValid {
                continue
            }
            right, rightValid := dfsOne(ingredients[1], visitedNodes)
            if !rightValid {
                continue
            }
            res.combinations = append(res.combinations, Recipe{
                left: left,
                right: right,
                total: element
            })
        }
    }
    return res, true
}

```

```

        if !rightValid {
            continue
        }
        res.combinations = append(res.combinations, Recipe{
            ingredient1: left,
            ingredient2: right,
        })
        return res, true
    }
}
return nil, false
}

```

### 3. Fungsi searchDFSMultiple(string, int) → ([]\*Tree, []int)

Fungsi ini bertugas mencari banyak jalur resep unik menuju elemen target dengan pendekatan DFS paralel menggunakan goroutine. Fungsi ini membuat konteks waktu agar proses tidak berjalan tanpa batas, lalu menjalankan pencarian untuk setiap kombinasi resep target menggunakan dfsSubTree. Untuk menghindari duplikasi struktur pohon, fungsi menggunakan serializeTree dan menyimpan hasil serialisasi dalam seenStructures. Hasilnya dikumpulkan dalam channel, dibatasi hingga sejumlah numOfPath, lalu dikembalikan sebagai slice dari objek Tree beserta jumlah elemen unik yang terlibat dalam tiap jalur.

```

func searchDFSMultiple(target string, numOfPath int) ([]*Tree,
[]int) {
    fmt.Println("start multiple")
    mainDataMul = recipeData.Recipes[target]

    ctx, cancel := context.WithTimeout(context.Background(),
30*time.Second)
    defer cancel()

    resultChan := make(chan *Node, numOfPath*2)
    var wg sync.WaitGroup

    var seenStructuresMutex sync.Mutex
    seenStructures := make(map[string]bool)

    targetCombs, exists := mainDataMul[target]
    if !exists || len(targetCombs) == 0 {
        return []*Tree{{root: &Node{element: target}}},
    []int{1}
    }
}

```

```

        for _, pair := range targetCombs {
            if len(pair) != 2 {
                continue
            }
            if pair[0] == target || pair[1] == target {
                fmt.Printf("loop detected: %s\n", target)
                continue
            }

            wg.Add(1)
            go func(combo []string) {
                defer wg.Done()
                currentPath := make(map[string]bool)
                currentPath[target] = true

                leftPath := copyVisitedMap(currentPath)
                leftResults := dfsSubTree(ctx, combo[0],
mainDataMul, leftPath, 0)

                if len(leftResults) == 0 {
                    return
                }
                rightPath := copyVisitedMap(currentPath)
                rightResults := dfsSubTree(ctx, combo[1],
mainDataMul, rightPath, 0)

                if len(rightResults) == 0 {
                    return
                }
                maxCombos := 3
                for i := 0; i < min(len(leftResults),
maxCombos); i++ {
                    for j := 0; j < min(len(rightResults),
maxCombos); j++ {
                        select {
                        case <-ctx.Done():
                            return
                        default:
                            finalTreeRoot := &Node{
                                element: target,
                                combinations:
[]Recipe{
                            ingredient1:
leftResults[i],
                            ingredient2:
rightResults[j],
                        },
                    }
                }
            }
        }
    }
}

```



```

        pathElementCounts = append(pathElementCounts,
getPathElementCount(rootNode))
    }

    fmt.Printf("found %d unique recipe paths\n", len(trees))
    return trees, pathElementCounts
}

```

4. Fungsi dfsSubTree(context.Context, string, map[string][][]string, map[string]bool, int) → []\*Node

Fungsi ini adalah DFS rekursif yang membentuk semua kemungkinan pohon resep dari suatu elemen secara mendalam dan mempertimbangkan berbagai kombinasi. Fungsi ini memiliki batas kedalaman maksimum, deteksi siklus melalui currentPath, serta menggunakan goroutine untuk paralelisme saat eksplorasi masih dangkal (depth rendah). Fungsi menggabungkan semua kombinasi valid dari bahan penyusun dan mengembalikan node-node hasil kombinasi sebagai alternatif jalur resep.

```

func dfsSubTree(ctx context.Context, element string,
currentRecipeMap map[string][][]string, currentPath
map[string]bool, depth int) []*Node {
    // fmt.Println(element)
    select {
    case <-ctx.Done():
        return []*Node{}
    default:
    }

    maxDepth := 15
    if depth > maxDepth {
        return []*Node{{element: element}}
    }

    if currentPath[element] {
        return []*Node{}
    }

    combs, exists := currentRecipeMap[element]
    if !exists || len(combs) == 0 || isBase(element) {
        return []*Node{{element: element}}
    }

    currentPath[element] = true

```

```

if depth < 3 && len(combs) > 1 {
    var wg sync.WaitGroup
    resultsMutex := sync.Mutex{}
    var allPossibleNodesForElement []*Node

    maxCombsToExplore := min(len(combs), 2)
    combsToExplore := combs[:maxCombsToExplore]

    for _, pair := range combsToExplore {
        if len(pair) != 2 {
            continue
        }

        wg.Add(1)
        go func(ingredients []string) {
            defer wg.Done()

            select {
            case <-ctx.Done():
                return
            default:
            }

            leftPath := copyVisitedMap(currentPath)
            leftResults := dfsSubTree(ctx,
            ingredients[0], currentRecipeMap, leftPath, depth+1)

            if len(leftResults) == 0 {
                return
            }

            rightPath :=
            copyVisitedMap(currentPath)
            rightResults := dfsSubTree(ctx,
            ingredients[1], currentRecipeMap, rightPath, depth+1)

            if len(rightResults) == 0 {
                return
            }

            var localNodes []*Node
            for i := 0; i < min(len(leftResults),
            2); i++ {
                for j := 0; j <
            min(len(rightResults), 2); j++ {
                    newNode := &Node{
                        element: element,

```

```

combinations:
    ingredient1:
    ingredient2:
    },
localNodes =
append(localNodes, newNode)

if len(localNodes) >= 3 {
    break
}
if len(localNodes) >= 3 {
    break
}
}

if len(localNodes) > 0 {
    resultsMutex.Lock()
    allPossibleNodesForElement =
append(allPossibleNodesForElement, localNodes...)
    resultsMutex.Unlock()
}
}(pair)
}

wg.Wait()

if len(allPossibleNodesForElement) > 5 {
    return allPossibleNodesForElement[:5]
}
return allPossibleNodesForElement
} else {
    var allPossibleNodesForElement []*Node

    maxCombsToExplore := 2
    if len(combs) > maxCombsToExplore {
        combs = combs[:maxCombsToExplore]
    }

    for _, pair := range combs {
        if len(pair) != 2 {
            continue
        }
    }
}

```

```

        leftPath := copyVisitedMap(currentPath)
        leftIngredientOptions := dfsSubTree(ctx,
pair[0], currentRecipeMap, leftPath, depth+1)
        if len(leftIngredientOptions) == 0 {
            continue
        }

        rightPath := copyVisitedMap(currentPath)
        rightIngredientOptions := dfsSubTree(ctx,
pair[1], currentRecipeMap, rightPath, depth+1)
        if len(rightIngredientOptions) == 0 {
            continue
        }

        maxOptions := 10
        leftLimit := min(len(leftIngredientOptions),
maxOptions)
        rightLimit :=
min(len(rightIngredientOptions), maxOptions)

        for i := 0; i < leftLimit; i++ {
            for j := 0; j < rightLimit; j++ {
                newNode := &Node{
                    element: element,
                    combinations: []Recipe{
                        ingredient1:
leftIngredientOptions[i],
                        ingredient2:
rightIngredientOptions[j],
                    },
                }
                allPossibleNodesForElement =
append(allPossibleNodesForElement, newNode)

                if len(allPossibleNodesForElement)
>= 5 {
                    return
                }
            }
        }
    }
}
return allPossibleNodesForElement
}
}

```

5. Fungsi  $\text{min}(\text{int}, \text{int}) \rightarrow \text{int}$

Fungsi pembantu sederhana ini mengembalikan nilai minimum dari dua bilangan bulat, digunakan untuk membatasi jumlah kombinasi atau opsi yang dieksplorasi selama pencarian.

```
func min(a, b int) int {
    if a < b {
        return a
    }
    return b
}
```

6. Prosedur  $\text{countUniqueElementsInPath}(*\text{Node}, \text{map}[\text{string}] \text{bool})$

Fungsi rekursif ini menelusuri pohon dari node tertentu dan menambahkan setiap elemen ke dalam map elements, yang digunakan untuk menghitung jumlah element yang terlibat dalam satu jalur resep.

```
func countUniqueElementsInPath(node *\text{Node}, elements
map[string]bool) {
    if node == nil {
        return
    }
    elements[node.element] = true
    if len(node.combinations) > 0 {
        recipe := node.combinations[0]
        countUniqueElementsInPath(recipe.ingredient1,
elements)
        countUniqueElementsInPath(recipe.ingredient2,
elements)
    }
}
```

7. Fungsi  $\text{getPathElementCount}(*\text{Node}) \rightarrow \text{int}$

Fungsi ini menghitung jumlah elemen unik dalam satu jalur pohon resep. Fungsi ini memanfaatkan  $\text{countUniqueElementsInPath}$  secara rekursif untuk menelusuri setiap node dalam pohon dan mencatat nama-nama elemen dalam map agar tidak terhitung dua kali.

```
func getPathElementCount(rootNode *\text{Node}) int {
    uniqueElements := make(map[string]bool)
    countUniqueElementsInPath(rootNode, uniqueElements)
    return len(uniqueElements)
}
```

8. Fungsi `copyVisitedMap(map[string]bool) → map[string]bool`

Fungsi ini membuat salinan dari map `currentPath` agar setiap jalur rekursi memiliki konteks path-nya sendiri. Ini penting untuk menghindari konflik antar goroutine saat membangun jalur resep secara paralel.

```
func copyVisitedMap(original map[string]bool) map[string]bool {
    copied := make(map[string]bool, len(original))
    for k, v := range original {
        copied[k] = v
    }
    return copied
}
```

9. Fungsi `serializeTree(*Node) → string`

Fungsi ini mengubah struktur pohon menjadi string unik yang merepresentasikan strukturnya. Ini digunakan untuk mendeteksi apakah struktur pohon tertentu sudah pernah ditemukan. Fungsi menyusun representasi elemen secara rekursif dan mengurutkan pasangan bahan dalam satu resep untuk menghindari duplikasi akibat urutan yang berbeda.

```
func serializeTree(node *Node) string {
    if node == nil {
        return ""
    }
    if len(node.combinations) == 0 {
        return node.element
    }
    left := serializeTree(node.combinations[0].ingredient1)
    right := serializeTree(node.combinations[0].ingredient2)

    if left > right {
        left, right = right, left
    }

    return fmt.Sprintf("%s(%s,%s)", node.element, left,
right)
}
```

## 4.4 Implementasi Back-end: Algoritma Bidirectional

### 4.4.1 Struktur Data

Selain menggunakan struktur data umum yang telah dijelaskan pada 4.1, berikut adalah struktur data lain yang digunakan dalam algoritma bidirectional.

Struktur Data	Penjelasan
Queue	Struktur data yang digunakan untuk menyimpan node apa saja yang perlu diproses dan telah diproses secara berurutan dengan logika FIFO (First In First Out)
Map	Struktur data yang digunakan untuk menyimpan informasi mengenai <i>predecessor</i> dari suatu Node yang telah dilalui. Hal ini penting untuk merekonstruksi kembali tree hasil yang merupakan jalur resep yang menjadi output.
Array	Struktur data ini banyak sekali digunakan yang secara umum untuk menyimpan beberapa elemen/data dengan tipe yang sama ke dalam satu kontainer.
Pair	Struktur data ini digunakan untuk menyimpan dua elemen yang merupakan kombinasi penyusun suatu elemen ke dalam satu kontainer. Dibutuhkan abstraksi untuk menganggap kedua Node penyusun menjadi satu kesatuan.

#### 4.4.2 Fungsi dan Prosedur

Berikut adalah fungsi dan prosedur yang digunakan.

1. Fungsi findBaseLeaves(node \*Nodebidir, baseLeaves []\*Nodebidir) []\*Nodebidir

Secara rekursif mencari semua node dalam sub-pohon yang berakar di node yang merupakan elemen dasar dan tidak memiliki anak

```
func findBaseLeaves(node *Nodebidir, baseLeaves []*Nodebidir)
[*Nodebidir] {
    if node == nil {
        return baseLeaves
    }
}
```

```

        if len(node.combinations) == 0 && isBase(node.element) {
            baseLeaves = append(baseLeaves, node)
        }

        for _, recipe := range node.combinations {
            if recipe.ingredient1 != nil &&
!recipe.ingredient1.isCycleNode {
                baseLeaves = findBaseLeaves(recipe.ingredient1,
baseLeaves)
            }
            if recipe.ingredient2 != nil &&
!recipe.ingredient2.isCycleNode {
                baseLeaves = findBaseLeaves(recipe.ingredient2,
baseLeaves)
            }
        }
        return baseLeaves
    }
}

```

2. Fungsi      bidirectionalSearchTree(tree            \*Treebidir,            recipesForItem  
map[string][][]string) \*Nodebidir

Fungsi inti untuk pencarian dua arah. Ia menginisialisasi dua queue: `q_f` untuk pencarian maju dari akar (`tree.root`) dan `q_b` untuk pencarian mundur dari semua `baseLeaves`. Pencarian dilakukan secara bergantian dari kedua arah. Saat sebuah node ditemukan oleh kedua pencarian (*overlapping*), `constructShortestPathTree` dipanggil untuk membuat pohon solusi.

```

func bidirectionalSearchTree(tree *Treebidir, recipesForItem
map[string][][]string) *Nodebidir {
    if tree == nil || tree.root == nil {
        fmt.Println("Tree is empty")
        return nil
    }
}

```

```

if tree.root.isCycleNode {
    fmt.Println("Root is a cycle node")
    return nil
}

q_f := list.New()
visited_f := make(map[*Nodebidir]*Nodebidir)
root_f := tree.root
q_f.PushBack(root_f)
visited_f[root_f] = nil

baseLeaves := findBaseLeaves(tree.root, []*Nodebidir{})
q_b := list.New()
visited_b := make(map[*Nodebidir]*Nodebidir)

if len(baseLeaves) == 0 {
    fmt.Println("No base leaves found, cannot perform backward
search.")
    return nil
}
for _, baseLeaf := range baseLeaves {
    q_b.PushBack(baseLeaf)
    visited_b[baseLeaf] = nil
}

forwardDepth := make(map[*Nodebidir]int)
backwardDepth := make(map[*Nodebidir]int)
baseLeafSource := make(map[*Nodebidir]*Nodebidir)

forwardDepth[root_f] = 0
for _, leaf := range baseLeaves {
    backwardDepth[leaf] = 0
    baseLeafSource[leaf] = leaf
}

for q_f.Len() > 0 && q_b.Len() > 0 {

```

```

        if q_f.Len() > 0 {
            frontElement_f := q_f.Front()
            curr_f_instance := frontElement_f.Value.(*Nodebidir)
            q_f.Remove(frontElement_f)

            if curr_f_instance.isCycleNode {
                continue
            }

            if _, found := backwardDepth[curr_f_instance]; found {
                pathTree :=
                constructShortestPathTree(curr_f_instance, visited_f, visited_b,
                recipesForItem)
                if pathTree != nil {
                    return pathTree
                } else {
                }
            }

            for _, recipe := range curr_f_instance.combinations {
                children := []*Nodebidir{recipe.ingredient1,
                recipe.ingredient2}
                for _, child_instance := range children {
                    if child_instance == nil ||
                    child_instance.isCycleNode {
                        continue
                    }
                    if _, v_found := visited_f[child_instance];
                    !v_found {
                        q_f.PushBack(child_instance)
                        visited_f[child_instance] =
                        curr_f_instance
                        forwardDepth[child_instance] =
                        forwardDepth[curr_f_instance] + 1
                        if _, b_found :=
                        backwardDepth[child_instance]; b_found {
                }
            }
        }
    }
}

```

```

pathTree :=
constructShortestPathTree(child_instance, visited_f, visited_b,
recipesForItem)
    if pathTree != nil {
        return pathTree
    } else {
        }
    }
}

if q_b.Len() > 0 {
    frontElement_b := q_b.Front()
    curr_b_instance := frontElement_b.Value.(*Nodebidir)
    q_b.Remove(frontElement_b)

    if curr_b_instance.isCycleNode {
        continue
    }

    if _, found := forwardDepth[curr_b_instance]; found {
        pathTree :=
constructShortestPathTree(curr_b_instance, visited_f, visited_b,
recipesForItem)
        if pathTree != nil {
            return pathTree
        } else {
        }
    }

    parent_instance := curr_b_instance.parent
    if parent_instance == nil ||
parent_instance.isCycleNode {
        continue
    }
}

```

```

    }
    if _, v_found := visited_b[parent_instance]; !v_found
    {
        q_b.PushBack(parent_instance)
        visited_b[parent_instance] = curr_b_instance
        currentBaseLeaf := baseLeafSource[curr_b_instance]
        baseLeafSource[parent_instance] = currentBaseLeaf
        backwardDepth[parent_instance] =
backwardDepth[curr_b_instance] + 1

        if _, f_found := forwardDepth[parent_instance];
f_found {
            pathTree :=
constructShortestPathTree(parent_instance, visited_f, visited_b,
recipesForItem)
            if pathTree != nil {
                return pathTree
            } else {
            }
        }
    }
    return nil
}

```

3. constructPathTree(meetingNode \*Nodebidir, visited\_f, visited\_b  
map[\*Nodebidir]\*Nodebidir, recipeData map[string][][]string) \*Nodebidir

Dipanggil ketika meetingNode (node pertemuan) ditemukan. Fungsi ini merekonstruksi jalur dari akar ke meetingNode menggunakan visited\_f (untuk jalur maju) dan dari meetingNode ke salah satu baseLeaf menggunakan visited\_b (untuk jalur mundur).

```

func constructShortestPathTree(meetingNode *Nodebidir, visited_f,
visited_b map[*Nodebidir]*Nodebidir, recipeData
map[string][][]string) *Nodebidir {
    forwardPath := []*Nodebidir{}
    curr := meetingNode
    for curr != nil {
        forwardPath = append([]*Nodebidir{curr}, forwardPath...)
        curr = visited_f[curr]
    }

    backwardPath := []*Nodebidir{}
    curr = meetingNode
    for curr != nil {
        backwardPath = append(backwardPath, curr)
        curr = visited_b[curr]
    }

    if len(backwardPath) > 0 {
        backwardPath = backwardPath[1:]
    }

    completePath := append(forwardPath, backwardPath...)
    return buildShortestPathTree(completePath, recipeData)
}

```

4. buildPathTree(path []\*Nodebidir, recipeData map[string][][]string) \*Nodebidir

Fungsi ini membuat *salinan* (clone) dari setiap node di path menggunakan nodeMap untuk melacak node asli ke salinannya. Kemudian, ia mengatur hubungan parent-child antar node salinan tersebut.

```

func buildShortestPathTree(path []*Nodebidir, recipeData
map[string][][]string) *Nodebidir {
    if len(path) == 0 {
        return nil
    }
}

```

```

    }

    nodeMap := make(map[*Nodebidir]*Nodebidir)

    for _, origNode := range path {
        nodeMap[origNode] = &Nodebidir{
            element:      origNode.element,
            combinations: []Recipebidir{},
        }
    }

    for i := 0; i < len(path)-1; i++ {
        origCurrent := path[i]
        origNext := path[i+1]

        nodeMap[origNext].parent = nodeMap[origCurrent]
    }

    expandNodeRecipes(path, nodeMap, recipeData)

    return nodeMap[path[0]]
}

```

5. `expandNodeRecipes(path []*Nodebidir, nodeMap map[*Nodebidir]*Nodebidir, recipeData map[string][][]string)`

Fungsi ini mengisi detail resep untuk setiap node yang merupakan bagian dari path (jalur terpendek yang sudah di-clone ke dalam nodeMap).

```

func expandNodeRecipes(path []*Nodebidir, nodeMap
map[*Nodebidir]*Nodebidir, recipeData map[string][][]string) {
    isOriginalNodeActuallyInPath := func(nodeToTest *Nodebidir,
    currentLinearPath []*Nodebidir) bool {
        if nodeToTest == nil {

```

```
        return false
    }
    for _, pathNode := range currentLinearPath {
        if pathNode == nodeToTest {
            return true
        }
    }
    return false
}

for _, origNode := range path {
    if isBase(origNode.element) {
        continue
    }

    clonedNode := nodeMap[origNode]

    if len(origNode.combinations) > 0 {
        bestRecipe := findBestRecipe(origNode, path)

        if (bestRecipe.ingredient1 == nil ||
bestRecipe.ingredient1.isCycleNode) &&
            (bestRecipe.ingredient2 == nil ||
bestRecipe.ingredient2.isCycleNode) {
            continue
        }

        ingredient1Cloned :=
createOrGetIngredientNode(bestRecipe.ingredient1, nodeMap,
clonedNode, path)
        ingredient2Cloned :=
createOrGetIngredientNode(bestRecipe.ingredient2, nodeMap,
clonedNode, path)

        if ingredient1Cloned != nil || ingredient2Cloned !=
nil {
```

```

        clonedNode.combinations =
append(clonedNode.combinations, Recipebidir{
            ingredient1: ingredient1Cloned,
            ingredient2: ingredient2Cloned,
        })
    }

    if ingredient1Cloned != nil &&
!isBase(ingredient1Cloned.element) &&
(bestRecipe.ingredient1 != nil &&
!isOriginalNodeActuallyInPath(bestRecipe.ingredient1, path)) {
        expandIngredientRecursively(ingredient1Cloned,
nodeMap, clonedNode, recipeData)
    }

    if ingredient2Cloned != nil &&
!isBase(ingredient2Cloned.element) &&
(bestRecipe.ingredient2 != nil &&
!isOriginalNodeActuallyInPath(bestRecipe.ingredient2, path)) {
        expandIngredientRecursively(ingredient2Cloned,
nodeMap, clonedNode, recipeData)
    }

} else if origNode != path[len(path)-1] {
    recipes, exists := recipeData[origNode.element]
    if exists && len(recipes) > 0 {
        bestRecipeStrings :=
findBestRecipeFromData(origNode.element, recipes, path)
        if len(bestRecipeStrings) == 2 {
            ing1Node := &Nodebidir{
                element: bestRecipeStrings[0],
                parent:  clonedNode,
            }

            ing2Node := &Nodebidir{
                element: bestRecipeStrings[1],

```

```
        parent: clonedNode,
    }

    clonedNode.combinations =
append(clonedNode.combinations, Recipebidir{
        ingredient1: ing1Node,
        ingredient2: ing2Node,
    })
    if !isBase(ing1Node.element) {
        expandIngredientRecursively(ing1Node,
nodeMap, clonedNode, recipeData)
    }
    if !isBase(ing2Node.element) {
        expandIngredientRecursively(ing2Node,
nodeMap, clonedNode, recipeData)
    }
}
}
```

6. findMultipleBidirectionalPaths(tree \*Treebidir, numPaths int) []\*Nodebidir

Berfungsi untuk mencari beberapa jalur resep sebanyak numPaths yang valid.

```
func findMultipleBidirectionalPaths(tree *Treebidir, numPaths int)
[]*Nodebidir {
    if tree == nil || tree.root == nil {
        fmt.Println("Tree is empty")
        return nil
    }

    if tree.root.isCycleNode {
```

```
        fmt.Println("Root is a cycle node")
        return nil
    }

    baseLeaves := findBaseLeaves(tree.root, []*Nodebidir{})
    if len(baseLeaves) == 0 {
        return nil
    }

    var validLeaves []*Nodebidir
    for _, leaf := range baseLeaves {
        if !leaf.isCycleNode {
            validLeaves = append(validLeaves, leaf)
        }
    }

    if len(validLeaves) == 0 {
        return nil
    }

    resultChan := make(chan PathResult, numPaths*2)
    var wg sync.WaitGroup
    var resultsMutex sync.Mutex
    var foundPaths []*Nodebidir
    var numFoundPaths int

    pathSignatures := make(map[string]bool)
    for i, singleBaseLeaf := range validLeaves {
        wg.Add(1)
        go func(leafIndex int, currentTargetLeaf *Nodebidir) {
            defer wg.Done()

            resultsMutex.Lock()
            shouldContinue := numFoundPaths < numPaths
            resultsMutex.Unlock()
```

```
        if !shouldContinue {
            return
        }
        if currentTargetLeaf.isCycleNode {
            return
        }

        path, score, desc, actualLeafFound :=
bidirectionalSearchFromLeaf(tree.root,
[]*Node{currentTargetLeaf})
        if path != nil && !isPathCyclic(path) {
            pathSignature := generatePathSignature(path)

            resultChan <- PathResult{
                path:           path,
                score:          score,
                desc:           desc,
                actualTargetLeaf: actualLeafFound,
                pathSignature:   pathSignature,
            }
        }
    }(i, singleBaseLeaf)
}

go func() {
    wg.Wait()
    close(resultChan)
}()

for result := range resultChan {
    resultsMutex.Lock()

    if isPathCyclic(result.path) {
        resultsMutex.Unlock()
        continue
    }
}
```

```
        if numFoundPaths < numPaths &&
!pathSignatures[result.pathSignature] {
    foundPaths = append(foundPaths, result.path)
    numFoundPaths++
    pathSignatures[result.pathSignature] = true
}
resultsMutex.Unlock()

if numFoundPaths >= numPaths {
    break
}
return foundPaths
}
```

7. bidirectionalSearchFromLeaf(root \*Nodebidir, targetBaseLeaves []\*Nodebidir) (\*Nodebidir, int, string, \*Nodebidir)

Fungsi ini melakukan pencarian dua arah antara root yang diberikan dan satu atau lebih targetBaseLeaves.

```
func bidirectionalSearchFromLeaf(root *Nodebidir, targetBaseLeaves []*Nodebidir) (*Nodebidir, int, string, *Nodebidir) {
    type MeetingPoint struct {
        node          *Nodebidir
        forwardDepth   int
        backwardDepth  int
        actualTargetLeaf *Nodebidir
    }

    if root == nil {
        return nil, -1, "", nil
    }
```

```
if len(targetBaseLeaves) == 0 {
    return nil, -1, "", nil
}

if root.isCycleNode {
    return nil, -1, "", nil
}

q_f := list.New()
visited_f := make(map[*Nodebidir]*Nodebidir)
q_f.PushBack(root)
visited_f[root] = nil
forwardDepth := make(map[*Nodebidir]int)
forwardDepth[root] = 0

q_b := list.New()
visited_b := make(map[*Nodebidir]*Nodebidir)
backwardDepth := make(map[*Nodebidir]int)
baseLeafSource := make(map[*Nodebidir]*Nodebidir)

validTargetsFound := false
for _, leaf := range targetBaseLeaves {
    if leaf == nil || leaf.isCycleNode {
        continue
    }
    q_b.PushBack(leaf)
    visited_b[leaf] = nil
    backwardDepth[leaf] = 0
    baseLeafSource[leaf] = leaf
    validTargetsFound = true
}

if !validTargetsFound {
    return nil, -1, "", nil
}
```

```

var meetingPoints []MeetingPoint

for q_f.Len() > 0 && q_b.Len() > 0 {
    currLevelSize_f := q_f.Len()
    for i := 0; i < currLevelSize_f; i++ {
        frontElement_f := q_f.Front()
        curr_f_instance := frontElement_f.Value.(*Nodebidir)
        q_f.Remove(frontElement_f)
        if curr_f_instance.isCycleNode {
            continue
        }

        if backDepth, found := backwardDepth[curr_f_instance];
        found {
            if actualLeaf, ok :=
            baseLeafSource[curr_f_instance]; ok {
                meetingPoints = append(meetingPoints,
                MeetingPoint{
                    node:           curr_f_instance,
                    forwardDepth:  forwardDepth[curr_f_instance],
                    backwardDepth: backDepth,
                    actualTargetLeaf: actualLeaf,
                })
            }
        }
    }

    for _, recipe := range curr_f_instance.combinations {
        children := []*Nodebidir{recipe.ingredient1,
        recipe.ingredient2}
        for _, child_instance := range children {
            if child_instance == nil ||
            child_instance.isCycleNode {
                continue
            }
            if _, v_found := visited_f[child_instance];

```

```

!v_found {
    q_f.PushBack(child_instance)
    visited_f[child_instance] =
curr_f_instance
    forwardDepth[child_instance] =
forwardDepth[curr_f_instance] + 1
    if backDepth, b_found :=
backwardDepth[child_instance]; b_found {
        if actualLeaf, ok :=
baseLeafSource[child_instance]; ok {
            meetingPoints =
append(meetingPoints, MeetingPoint{
                node:
child_instance,
                forwardDepth:
forwardDepth[child_instance],
                backwardDepth: backDepth,
                actualTargetLeaf: actualLeaf,
            })
        }
    }
}
}
}
}
}
currLevelSize_b := q_b.Len()
for i := 0; i < currLevelSize_b; i++ {
    frontElement_b := q_b.Front()
    curr_b_instance := frontElement_b.Value.(*Nodebidir)
    q_b.Remove(frontElement_b)
    if curr_b_instance.isCycleNode {
        continue
    }

    if fwdDepth, found := forwardDepth[curr_b_instance];
found {

```

```

        if actualLeaf, ok :=
baseLeafSource[curr_b_instance]; ok {
            meetingPoints = append(meetingPoints,
MeetingPoint{
                node:           curr_b_instance,
                forwardDepth:   fwdDepth,
                backwardDepth:
backwardDepth[curr_b_instance],
                actualTargetLeaf: actualLeaf,
            })
        }
    }

    parent_instance := curr_b_instance.parent
    // Skip nil or cyclic parent nodes entirely
    if parent_instance == nil ||
parent_instance.isCycleNode {
        continue
    }
    if _, v_found := visited_b[parent_instance]; !v_found
{
    q_b.PushBack(parent_instance)
    visited_b[parent_instance] = curr_b_instance
    backwardDepth[parent_instance] =
backwardDepth[curr_b_instance] + 1
    if sourceLeaf, ok :=
baseLeafSource[curr_b_instance]; ok { // Propagate source
        baseLeafSource[parent_instance] = sourceLeaf
        if fwdDepth, f_found :=
forwardDepth[parent_instance]; f_found {
            meetingPoints = append(meetingPoints,
MeetingPoint{
                node:           parent_instance,
                forwardDepth:   fwdDepth,
                backwardDepth:
backwardDepth[parent_instance],
            })
        }
    }
}

```

```
        actualTargetLeaf: sourceLeaf,
    })
}
}
}

if len(meetingPoints) > 0 {
    break
}
}

if len(meetingPoints) == 0 {
    return nil, -1, "", nil
}

var bestMeetingPointData MeetingPoint
foundBest := false
bestTotalDepth := -1

for _, mp := range meetingPoints {
    totalDepth := mp.forwardDepth + mp.backwardDepth
    if !foundBest || totalDepth < bestTotalDepth {
        bestTotalDepth = totalDepth
        bestMeetingPointData = mp
        foundBest = true
    } else if totalDepth == bestTotalDepth {
        if bestMeetingPointData.actualTargetLeaf != nil &&
mp.actualTargetLeaf != nil && // Ensure not nil
            mp.backwardDepth <
bestMeetingPointData.backwardDepth {
            bestMeetingPointData = mp
        }
    }
}
```

```

        if !foundBest || bestMeetingPointData.actualTargetLeaf == nil
    {
        return nil, -1, "", nil
    }

    pathDesc := fmt.Sprintf("Path to %s (total depth: %d)",
        bestMeetingPointData.actualTargetLeaf.element,
        bestTotalDepth)

    recipesMapConverted :=
        map[string][][]string(recipeDatas.Recipes)
    resultTree :=
        constructShortestPathTree(bestMeetingPointData.node, visited_f,
        visited_b, recipesMapConverted)
    if isPathCyclic(resultTree) {
        return nil, -1, "", nil
    }

    return resultTree, bestTotalDepth, pathDesc,
        bestMeetingPointData.actualTargetLeaf
}

```

## 4.5 Implementasi Back-end: Server dan Data Scraping

Di bawah ini adalah struktur data dan fungsi atau prosedur yang digunakan untuk kepentingan server dan *data scraping*.

### 4.5.1 Struktur Data

Struktur Data	Penjelasan
<b>OutputData</b> <code>type OutputData struct {     Elements []string</code>	Struktur data penyimpanan hasil scraping yang tersimpan dalam json. Elements adalah daftar elemen yang ada sedangkan Recipes berisikan resep pembentuk per elemennya.

```

`json:"elements"`
    Recipes
map[string]map[string][][]string
`json:"recipes"`
}

```

#### 4.5.2 Prosedur dan Fungsi

1. ScrapeRecipes() (OutputData, error)

Fungsi utama yang melakukan proses *scraping* (pengambilan data) dari halaman web Little Alchemy 2 Fandom Wiki. Menggunakan goquery untuk mem-parsing HTML dari respons. Logika pemilihan target scraping menggunakan iterasi melalui setiap elemen <p> (paragraf) di dokumen HTML. Jika sebuah paragraf berisi teks yang mengindikasikan awal dari tabel resep untuk tier tertentu (misalnya, "These elements can be created by combining only..."), maka ia akan memproses tabel (<table>) berikutnya.

```

func ScrapeRecipes() (OutputData, error) {
    // Initialize the result structure with starting elements
    result := OutputData{
        Elements: []string{"Air", "Earth", "Fire", "Water"},
        Recipes:  make(map[string]map[string][][]string),
    }

    // Define base elements
    baseElements := map[string]bool{
        "Air":   true,
        "Earth": true,
        "Fire":  true,
        "Water": true,
    }

    // Define the elements to exclude
    excludedElements := map[string]bool{
        "Time": true,
    }
}

```

```
        "Ruins":         true,
        "Archeologist": true,
    }

    // Track available elements (initially just base elements)
    availableElements := map[string]bool{
        "Air":   true,
        "Earth": true,
        "Fire":  true,
        "Water": true,
    }

    res, err := http.Get(url)
    if err != nil {
        return OutputData{}, err
    }
    defer res.Body.Close()
    if res.StatusCode != http.StatusOK {
        return OutputData{}, fmt.Errorf("status code error: %d %s", res.StatusCode, res.Status)
    }

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

    // Store all recipes we find
    allRecipes := make(map[string][][]string)

    // Process tables in order (which handles tiers implicitly)
    doc.Find("p").Each(func(_ int, p *goquery.Selection) {
        if strings.Contains(p.Text(), "These elements can be created by combining only") {
            nextTable := p.NextFiltered("table")
            if nextTable.Length() > 0 {
```

```

        // Elements in this table/tier
        elementsInCurrentTable := []string{}

        // Extract data from this table
        rows := nextTable.Find("tr").Slice(1,
nextTable.Find("tr").Length())
        rows.Each(func(_ int, row *goquery.Selection) {
            cols := row.Find("td")
            if cols.Length() == 2 {
                // Get element name
                elementName, exists :=
cols.Eq(0).Find("a").Attr("title")
                if !exists || elementName == "" {
                    elementName =
strings.TrimSpace(cols.Eq(0).Find("a").Text())
                }

                if elementName == "" {
                    return
                }

                // Skip excluded elements
                if excludedElements[elementName] {
                    fmt.Printf("Skipping excluded element:
%s\n", elementName)
                    return
                }

                // Track this element for this table
                elementsInCurrentTable =
append(elementsInCurrentTable, elementName)

                // Extract all recipes for this element
                elementRecipes := [][]string{}
                cols.Eq(1).Find("li").Each(func(_ int, li
*goquery.Selection) {

```

```
recipe := []string{}
containsExcluded := false

li.Find("a").Each(func(_ int, a
*goquery.Selection) {
    text := a.Text()
    if text == "" {
        return
    }

    ingredient, exists :=
a.Attr("title")
    if !exists || ingredient == "" {
        ingredient =
strings.TrimSpace(text)
    }

    // Check if this is an excluded
ingredient
    if excludedElements[ingredient] {
        containsExcluded = true
        return
    }

    if ingredient != "" {
        recipe = append(recipe,
ingredient)
    }
})

// Only add recipes that don't contain
excluded elements
if len(recipe) > 0 &&
!containsExcluded {
    elementRecipes =
append(elementRecipes, recipe)
```

```

        }
    })

        if len(elementRecipes) > 0 {
            allRecipes[elementName] =
elementRecipes
        }
    }

// Now process elements in this table
for _, element := range elementsInCurrentTable {
    recipes, exists := allRecipes[element]
    if !exists || len(recipes) == 0 {
        continue
    }

        // Filter recipes to only include those with
available ingredients
        validRecipes := [][]string{}
        for _, recipe := range recipes {
            valid := true
            for _, ingredient := range recipe {
                if !availableElements[ingredient] {
                    valid = false
                    break
                }
            }

            if valid {
                validRecipes = append(validRecipes,
recipe)
            }
        }

        // If we have valid recipes, add this element
    }
}

```

```

        to result
            if len(validRecipes) > 0 {
                // Add to elements list and mark as
                available
                result.Elements = append(result.Elements,
                element)
                availableElements[element] = true

                // Initialize recipe map for this element
                result.Recipes[element] =
                make(map[string][][]string)

                // Add the valid recipes for this element
                result.Recipes[element][element] =
                validRecipes

                // Now recursively add recipes for all
                non-base ingredients
                addedIngredients := make(map[string]bool)
                // Avoid duplicates
                addRecipesRecursively(element,
                validRecipes, result, baseElements, addedIngredients)

                fmt.Printf("Added element %s with %d valid
                recipes\n", element, len(validRecipes))
            }
        }
    }
}

fmt.Printf("Total elements: %d, Total elements with recipes:
%d\n",
len(result.Elements), len(result.Recipes))

return result, nil

```

```
}
```

2. addRecipesRecursively(targetElement string, recipes [][]string, result OutputData, baseElements map[string]bool, addedIngredients map[string]bool)

Prosedur rekursif yang bertujuan untuk "melengkapi" data resep.

```
func addRecipesRecursively(targetElement string, recipes
    [][]string, result OutputData,
    baseElements map[string]bool, addedIngredients
    map[string]bool) {
    for _, recipe := range recipes {
        for _, ingredient := range recipe {
            // Skip base elements and already processed
            ingredients for this target
            if baseElements[ingredient] ||
            addedIngredients[ingredient] {
                continue
            }

            // Mark this ingredient as processed for this target
            addedIngredients[ingredient] = true

            // Find this ingredient's recipes in the result
            if ingredientRecipesMap, exists :=
            result.Recipes[ingredient]; exists {
                if ingredientRecipes, exists :=
                ingredientRecipesMap[ingredient]; exists {
                    // Add this ingredient's recipes to the target
                    element
                    result.Recipes[targetElement][ingredient] =
                    ingredientRecipes

                    // Recursively add this ingredient's
                }
            }
        }
    }
}
```

```
    ingredients' recipes
        addRecipesRecursively(targetElement,
    ingredientRecipes, result, baseElements, addedIngredients)
    }
}
}
}
```

3. SaveRecipesToJson(data OutputData, filename string) error

Prosedur utilitas yang menyimpan data resep yang telah di-scrape (dalam format OutputData) ke sebuah file JSON.

```
func SaveRecipesToJson(data OutputData, filename string) error {
    jsonData, err := json.MarshalIndent(data, "", " ")
    if err != nil {
        return err
    }

    err = os.WriteFile(filename, jsonData, 0644)
    if err != nil {
        return err
    }

    return nil
}
```

4. searchHandler(w http.ResponseWriter, r \*http.Request)

Prosedur yang bertugas untuk memproses panggilan dari frontend dan mengembalikan respon yang tepat sesuai permintaan.

```
func searchHandler(w http.ResponseWriter, r *http.Request) {
    log.Println("Received search request")

    w.Header().Set("Content-Type", "application/json")
    w.Header().Set("Access-Control-Allow-Origin", "*")
    w.Header().Set("Access-Control-Allow-Methods", "POST,
OPTIONS")
    w.Header().Set("Access-Control-Allow-Headers", "Content-Type")

    if r.Method == "OPTIONS" {
        w.WriteHeader(http.StatusOK)
        return
    }

    if r.Method != "POST" {
        http.Error(w, `{"error":"method not allowed"}`,
http.StatusMethodNotAllowed)
        log.Printf("Method not allowed: %s\n", r.Method)
        return
    }

    var req SearchRequest
    if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
        http.Error(w, `{"error":"invalid input"}`,
http.StatusBadRequest)
        log.Printf("Failed to decode request: %v\n", err)
        return
    }

    log.Printf("Searching for target: '%s' using algorithm: %s,
mode: %s, maxRecipes: %d\n",
        req.Target, req.Algorithm, req.SearchMode, req.MaxRecipes)

    target := req.Target
    fmt.Printf("Target: %s\n", target)
```

```

startTime := time.Now()

var resp interface{}
if req.SearchMode == "single" {
    var node int
    var tree *Tree
    if req.Algorithm == "DFS" {
        tree, node = searchDFSOne(target)
        treeNode := convertToTreeNode(tree.root)
        executionTime := time.Since(startTime).Milliseconds()
        resp = SearchResponse{
            Trees:          []*TreeNode{treeNode},
            ExecutionTime: float64(executionTime),
            NodesVisited:  []int{node},
        }
    }

} else if req.Algorithm == "BFS" {
    fmt.Println("tes")
    tree, node := searchBFSOne(target)
    treeNode := convertToTreeNode(tree.root)
    executionTime := time.Since(startTime).Milliseconds()
    resp = SearchResponse{
        Trees:          []*TreeNode{treeNode},
        ExecutionTime: float64(executionTime),
        NodesVisited:  []int{node},
    }
}

} else {
    tree := searchBidirectOne(target)
    fmt.Println("tes")
    treeNode := convertToTreeNode2(tree)
    executionTime := time.Since(startTime).Milliseconds()
    resp = SearchResponse{
        Trees:          []*TreeNode{treeNode},
        ExecutionTime: float64(executionTime),
        NodesVisited:  []int{node},
    }
}

```

```

        }

    }

} else { // multiple
    var trees []*Tree
    var nodeVisited []int
    if req.Algorithm == "DFS" {
        maxRecipes := req.MaxRecipes
        if maxRecipes <= 1 {
            var node int
            var tree *Tree
            tree, node = searchDFSOne(target)
            treeNode := convertToTreeNode(tree.root)
            executionTime :=
time.Since(startTime).Milliseconds()
            resp = SearchResponse{
                Trees:          []*TreeNode{treeNode},
                ExecutionTime: float64(executionTime),
                NodesVisited:  []int{node},
            }
        } else {
            trees, nodeVisited = searchDFSMultiple(target,
maxRecipes)
            var treeNodes []*TreeNode
            for _, tree := range trees {
                treeNode := convertToTreeNode(tree.root)
                treeNodes = append(treeNodes, treeNode)
            }

            executionTime :=
time.Since(startTime).Milliseconds()
            resp = MultipleSearchResponse{
                Trees:          treeNodes,
                ExecutionTime: float64(executionTime),
                NodesVisited:  nodeVisited,
            }
        }
    }
}

```

```

    } else if req.Algorithm == "BFS" {
        maxRecipes := req.MaxRecipes
        if maxRecipes <= 1 {
            tree, node := searchBFSOne(target)
            treeNode := convertToTreeNode(tree.root)
            executionTime :=
time.Since(startTime).Milliseconds()
            resp = SearchResponse{
                Trees:          []*TreeNode{treeNode},
                ExecutionTime: float64(executionTime),
                NodesVisited:  []int{node},
            }
        } else {
            trees, nodeVisited := searchBFSMultiple(target,
maxRecipes) //changed to check
            var treeNodes []*TreeNode
            for _, tree := range trees {
                treeNode := convertToTreeNode(tree.root)
                treeNodes = append(treeNodes, treeNode)
            }

            executionTime :=
time.Since(startTime).Milliseconds()
            resp = MultipleSearchResponse{
                Trees:          treeNodes,
                ExecutionTime: float64(executionTime),
                NodesVisited:  nodeVisited,
            }
        }
    } else {
        maxRecipes := req.MaxRecipes
        if maxRecipes <= 0 {
            maxRecipes = 1 // Default value
        }
        trees := searchBidirectionMultiple(target, maxRecipes)
        var treeNodes []*TreeNode
    }
}

```

```

        for _, tree := range trees {
            treeNode := convertToTreeNode2(tree)
            treeNodes = append(treeNodes, treeNode)
        }

        executionTime := time.Since(startTime).Milliseconds()

        // Define a new response structure for multiple trees
        type MultipleSearchResponse struct {
            Trees          []*TreeNode `json:"trees"`
            NodesVisited  int          `json:"nodesVisited"`
            ExecutionTime float64      `json:"executionTime"`
        }

        // Calculate total nodes visited if multiple counts
        // were returned
        totalNodes := 0
        if len(nodeVisited) > 0 {
            for _, n := range nodeVisited {
                totalNodes += n
            }
        }

        resp = MultipleSearchResponse{
            Trees:          treeNodes,
            ExecutionTime: float64(executionTime),
            NodesVisited:  totalNodes,
        }
    }
}

respData, err := json.Marshal(resp)
if err != nil {
    http.Error(w, `{"error":"internal server error"}`, http.StatusInternalServerError)
    log.Printf("Failed to marshal response: %v\n", err)
    return
}

```

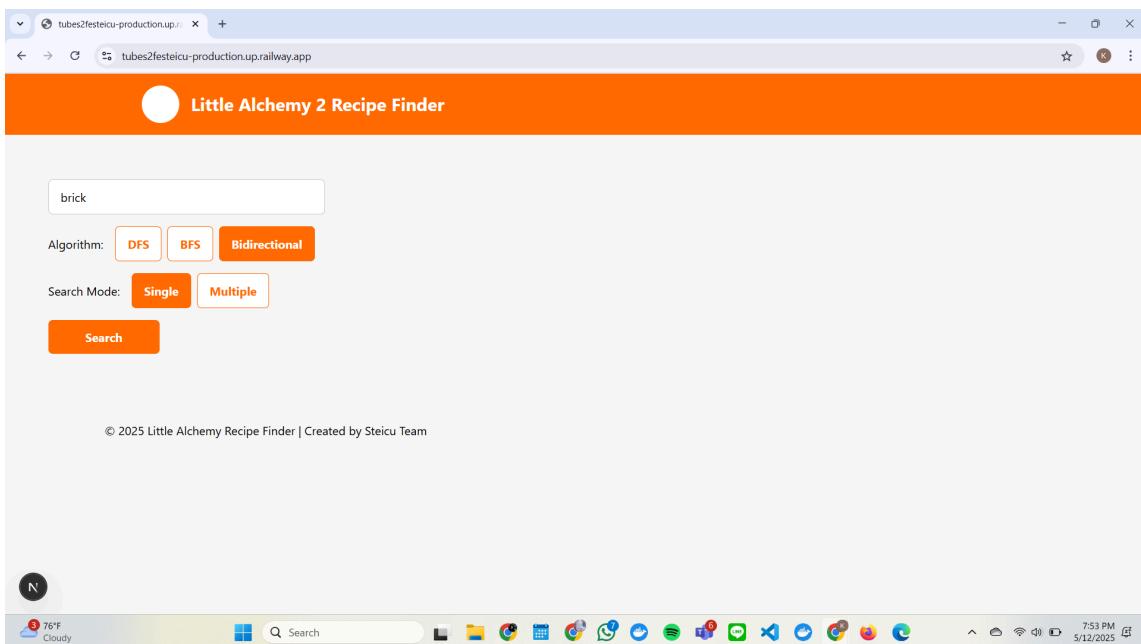
```
    }

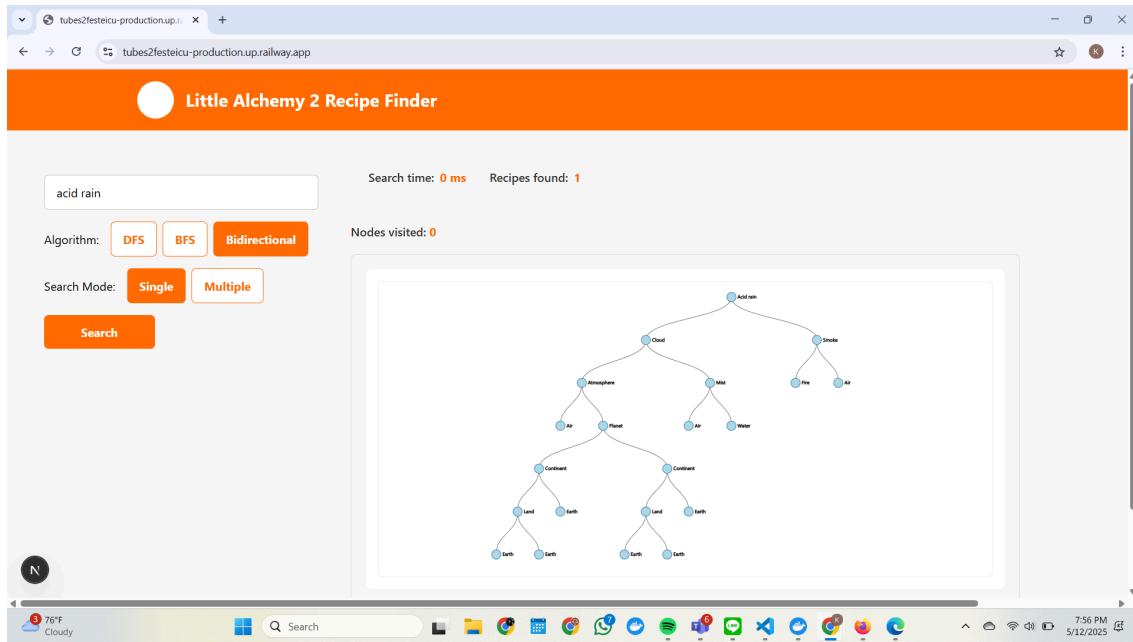
    log.Printf("Sending response: %s\n", string(respData))
    w.WriteHeader(http.StatusOK)
    w.Write(respData)
}
```

## 4.6 Tata Cara Penggunaan Program

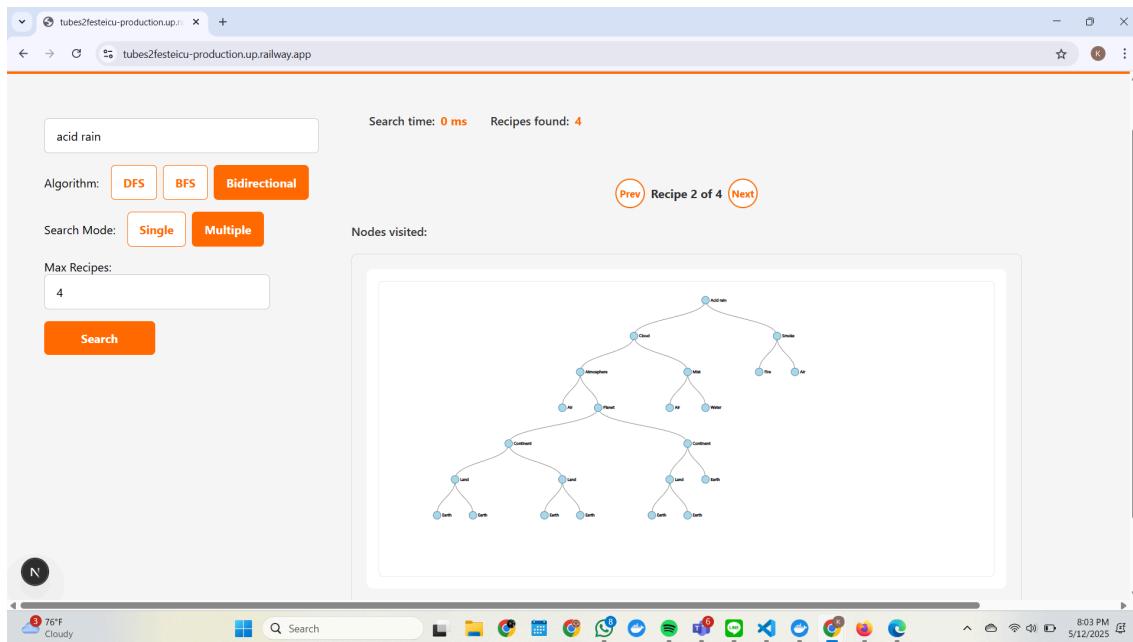
### 4.6.1 Laman Pencarian

Laman pada website projek ini hanya terdiri dari satu laman yang berisi semua informasi utama program STEIcu. Pada laman ini, pengguna menginputkan nama elemen bisa dengan diawali huruf kapital maupun tidak. Pada laman ini juga tersedia pilihan bagi pengguna untuk mencari resep elemen dengan algoritma BFS, DFS, atau Bidirectional. Pengguna dapat menekan pilihannya sesuai tulisan yang ada. Setelah itu, pengguna dapat menentukan jumlah resep yang ingin dicari. Apabila hanya menginginkan memunculkan satu resep, pengguna dapat menekan tombol Single. Apabila menginginkan lebih dari satu maka pengguna dapat memilih tombol Multiple, lalu pengguna akan menentukan berapa jumlah resep yang diinginkan dengan minimal satu. Selanjutnya, pengguna dapat menekan tombol Search untuk memunculkan resep dengan algoritma yang sesuai.



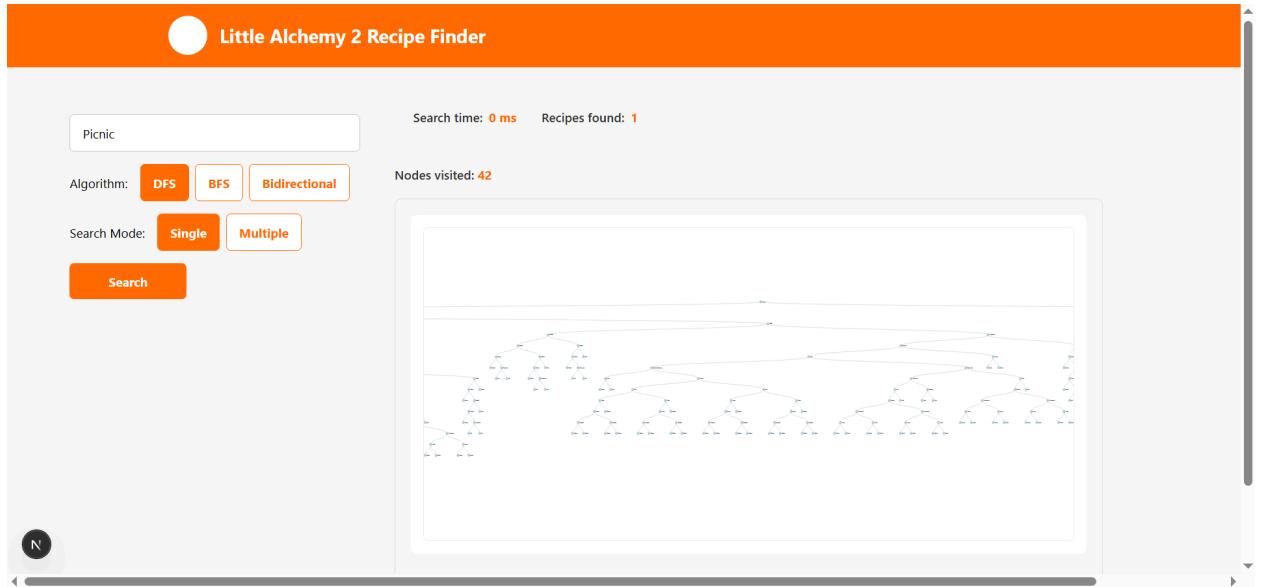


Hasil resep berbentuk tree ditampilkan di sebelah kanan pencarian, gambar diatas adalah contoh tampilan single. Saat memilih multiple, pengguna dapat melihat berbagai kombinasi resep dengan menekan tombol Next untuk ke resep selanjutnya dan tombol Prev untuk ke resep sebelumnya. Berikut tampilan Multiple:



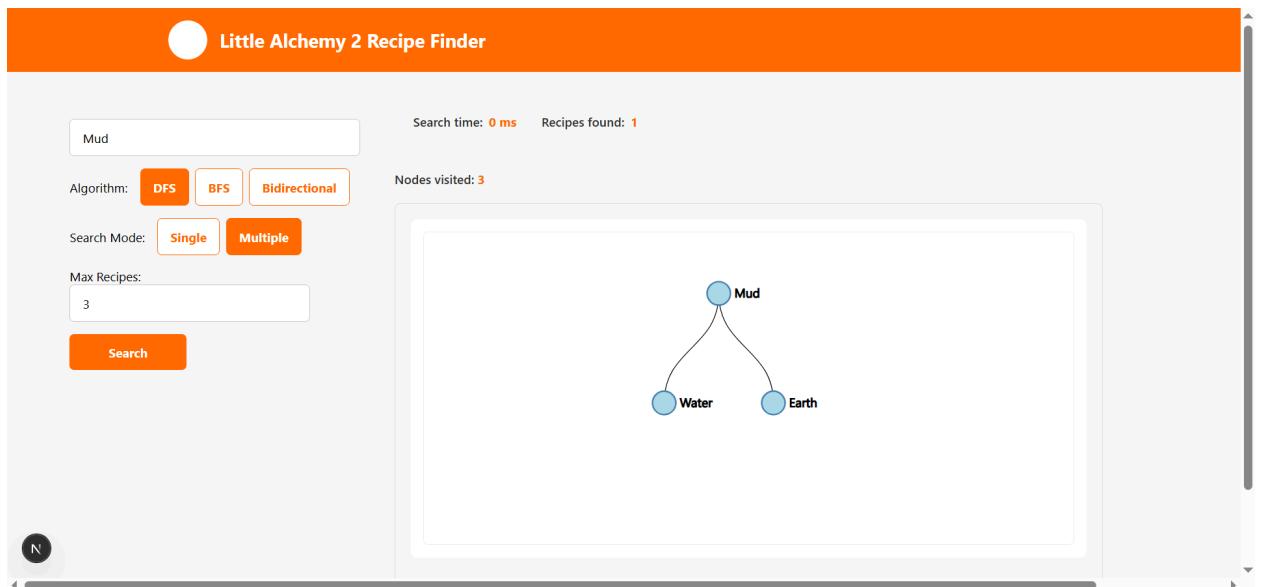
## 4.7 Hasil Pengujian dan Analisis

### 4.7.1 Test case 1: DFS Single Recipe



Hasil uji pencarian elemen Picnic (tier 15)

### 4.7.2. Tect case 2: DFS Multiple Recipe



Hasil uji pencarian elemen Mud yang hanya memiliki satu resep

#### 4.7.3 Test case 3: BFS Single Recipe

**Little Alchemy 2 Recipe Finder**

Meat

Search time: 0 ms Recipes found: 1

Nodes visited: 152

Algorithm:  DFS  BFS  Bidirectional

Search Mode:  Single  Multiple

Search

N

Hasil uji pencarian elemen Meat (tier 8)

#### 4.7.4 Test case 4: BFS Multiple Recipe

**Little Alchemy 2 Recipe Finder**

Dam

Search time: 4 ms Recipes found: 3

Nodes visited: 7

Algorithm:  DFS  BFS  Bidirectional

Search Mode:  Single  Multiple

Max Recipes: 3

Search

Prev Recipe 1 of 3 Next

N

Hasil uji pencarian 3 resep elemen Dam (tier 4)

Little Alchemy 2 Recipe Finder

Dam

Search time: 4 ms Recipes found: 3

Algorithm: **BFS** DFS Bidirectional

Search Mode: Single Multiple

Max Recipes: 3

Nodes visited: 7

Recipe 2 of 3

```
graph TD; Dam --> Water; Dam --> Wall; Water --> Stone1[Stone]; Water --> Pressure1[Pressure]; Stone1 --> Earth1[Earth]; Stone1 --> Air1[Air]; Pressure1 --> Air2[Air]; Pressure1 --> Air3[Air]; Wall --> Stone2[Stone]; Wall --> Pressure2[Pressure]; Stone2 --> Earth2[Earth]; Stone2 --> Air4[Air]; Pressure2 --> Lava[Lava]; Pressure2 --> Fire[Fire]
```

Hasil uji pencarian 3 resep elemen Dam (tier 4)

Little Alchemy 2 Recipe Finder

Dam

Search time: 4 ms Recipes found: 3

Algorithm: **BFS** DFS Bidirectional

Search Mode: Single Multiple

Max Recipes: 3

Nodes visited: 9

Recipe 3 of 3

```
graph TD; Dam --> Water; Dam --> Wall; Water --> Stone1[Stone]; Water --> Pressure1[Pressure]; Stone1 --> Earth1[Earth]; Stone1 --> Air1[Air]; Pressure1 --> Air2[Air]; Pressure1 --> Air3[Air]; Wall --> Stone2[Stone]; Wall --> Pressure2[Pressure]; Stone2 --> Earth2[Earth]; Stone2 --> Air4[Air]; Pressure2 --> Lava[Lava]; Pressure2 --> Fire[Fire]
```

Hasil uji pencarian 3 resep elemen Dam (tier 4)

#### 4.7.5 Test case 5: Single Bidirectional

Astronomer

Search time: 4 ms   Recipes found: 1

Algorithm:

Nodes visited: 0

Search Mode:

**Search**

N

Hasil uji pencarian resep elemen Astronomer (tier 8)

#### 4.7.6 Test case 6: Multiple Bidirectional

Human

Search time: 1991 ms   Recipes found: 2

Algorithm:

Search Mode:

Max Recipes: 2

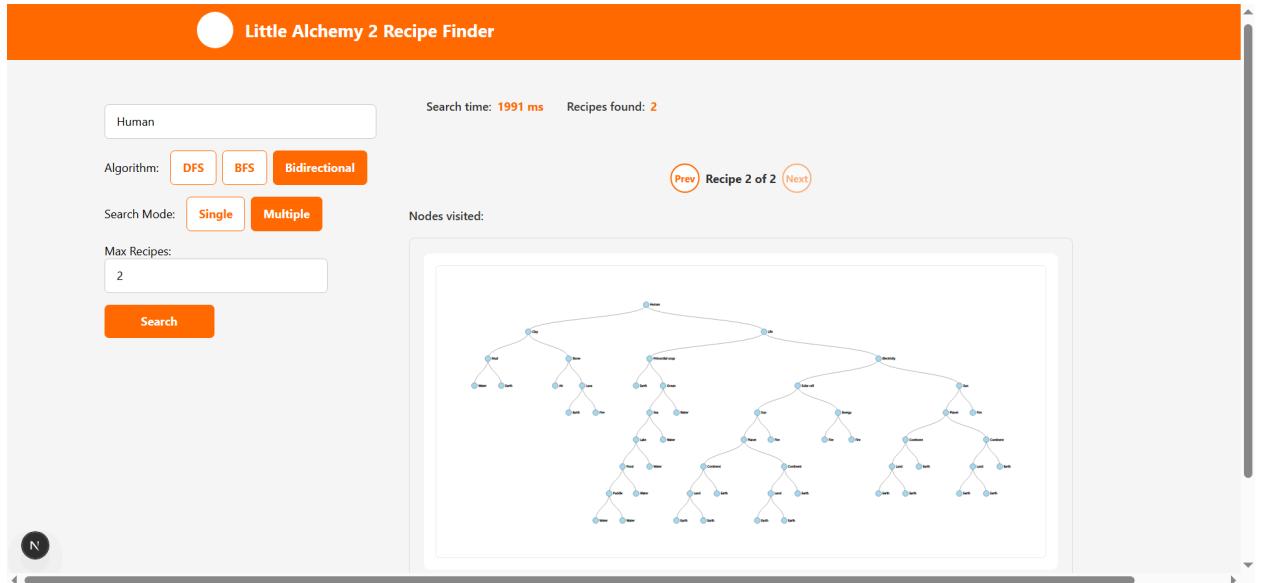
**Search**

Nodes visited:

Prev Recipe 1 of 2 Next

N

Hasil uji pencarian 2 resep elemen Human (tier 7)



Hasil uji pencarian 2 resep elemen Human (tier 7)

#### 4.7.7 Analisis Hasil Pengujian

Setelah melakukan cukup banyak pengujian yang dilakukan di luar apa yang tertera dalam laporan, terdapat beberapa wawasan yang dapat diambil. Pertama, untuk mencari satu resep yang pertama kali muncul, pencarian melalui pendekatan DFS secara umum lebih cepat dibandingkan pencarian melalui pendekatan BFS. Hal ini dapat terjadi karena metode DFS pada dasarnya langsung hanya berfokus untuk mencari satu subpohon/kombinasi resep saja tanpa mengeksplor simpul di subpohon lain. Namun, untuk beberapa kasus, bisa saja pencarian satu resep menggunakan BFS lebih cepat dari DFS. Hal ini dapat terjadi karena terdapat kasus panjang dari cabang pertama (satu-satunya cabang yang ditelusuri oleh algoritma DFS) secara signifikan jauh lebih panjang dari cabang lain, menyebabkan algoritma BFS bisa berhenti lebih cepat.

Wawasan yang kedua adalah dalam skenario pencarian beberapa resep, metode pencarian melalui pendekatan BFS secara umum lebih cepat dibandingkan pencarian melalui pendekatan DFS. Hal ini dapat terjadi karena penggunaan *multithreading* sehingga jumlah cabang yang ada tidak terlalu memengaruhi kecepatan pencarian, yang memengaruhi hanyalah kedalaman cabang. Menggunakan BFS, algoritma lebih mungkin untuk berhenti lebih cepat karena akan langsung berhenti pada n cabang terpendek yang ada di pohon tersebut.

Wawasan yang ketiga adalah *bidirectional searching* adalah bentuk optimalisasi dari metode BFS yang dalam kasus pohon resep yang sangat besar, berhasil memangkas waktu pencarian BFS secara cukup signifikan. Hal ini terjadi karena pada dasarnya *bidirectional searching* membuat jumlah simpul yang telah dilalui dua kali lipat dibandingkan BFS biasa dalam jangka waktu yang sama. Terlebih jika dipadukan dengan *multithreading*, algoritma ini adalah algoritma yang secara signifikan lebih baik dari BFS biasa terutama dalam konteks kompleksitas waktu, walaupun tentunya dari segi memori cukup ada pengorbanan yang harus dibuat.

## **BAB V**

### **KESIMPULAN DAN SARAN**

#### **5.1 Kesimpulan**

Aplikasi “Little Alchemy 2 Recipe Finder” berhasil dibuat sebagai alat bantu untuk mencari dan menampilkan kombinasi elemen dalam game Little Alchemy 2. Aplikasi ini dibangun dengan antarmuka modern menggunakan Next.js di bagian depan dan Golang di bagian belakang, sehingga tampilannya nyaman digunakan dan performanya cepat. Untuk mencari jalur pembuatan elemen, digunakan algoritma seperti BFS dan DFS yang bisa menelusuri kombinasi secara efisien. Selain itu, kami juga berhasil menyelesaikan fitur bonus berupa pencarian dua arah (bidirectional), yang membuat proses pencarian jadi lebih cepat karena dilakukan dari dua sisi sekaligus. Di sisi backend, kami menambahkan fitur multithreading supaya bisa mencari beberapa elemen sekaligus secara paralel, serta scraping otomatis agar data elemen selalu lengkap dan terbaru. Visualisasi pohon kombinasi yang ditampilkan juga memudahkan pengguna untuk melihat langkah-langkah pembuatan elemen secara menyeluruh. Secara keseluruhan, alat ini sudah bisa digunakan dengan nyaman dan memberikan pengalaman eksplorasi kombinasi elemen yang seru dan informatif.

#### **5.2 Saran**

Untuk ke depannya, aplikasi ini bisa ditingkatkan lagi dengan menambahkan algoritma yang lebih pintar, seperti A\*, supaya pencarian bisa lebih cepat lagi terutama untuk elemen-elemen rumit. Visualisasi pohon dapat dibuat lebih interaktif, misalnya dengan fitur zoom, sorot elemen tertentu, atau animasi langkah demi langkah. Menambahkan kemampuan untuk mencari beberapa elemen sekaligus juga akan sangat berguna bagi pengguna yang ingin membuat banyak kombinasi dalam satu waktu. Kalau memungkinkan, bisa juga dibuat versi offline supaya tetap bisa dipakai meskipun tidak ada internet. Dengan beberapa pengembangan tambahan, alat ini bisa jadi teman main yang asyik dan membantu banget buat para pemain Little Alchemy 2.

## LAMPIRAN

- Tautan Repository GitHub :

Frontend : [https://github.com/wrdtlkhoir/Tubes2\\_FE\\_STEIcu.git](https://github.com/wrdtlkhoir/Tubes2_FE_STEIcu.git)

Backend : [https://github.com/wrdtlkhoir/Tubes2\\_BE\\_STEIcu.git](https://github.com/wrdtlkhoir/Tubes2_BE_STEIcu.git)

- Tautan Video Youtube :  IF2211 Strategi Algoritma - Tugas Besar 2

No	Poin	Ya	Tidak
1	Aplikasi dapat dijalankan.	v	
2	Aplikasi dapat memperoleh data <i>recipe</i> melalui scraping.	v	
3	Algoritma <i>Depth First Search</i> dan <i>Breadth First Search</i> dapat menemukan <i>recipe</i> elemen dengan benar.	v	
4	Aplikasi dapat menampilkan visualisasi <i>recipe</i> elemen yang dicari sesuai dengan spesifikasi.	v	
5	Aplikasi mengimplementasikan multithreading.	v	
6	Membuat laporan sesuai dengan spesifikasi.	v	
7	Membuat bonus video dan diunggah pada Youtube.	v	
8	Membuat bonus algoritma pencarian <i>Bidirectional</i> .	v	

9	Membuat bonus <i>Live Update</i> .		v
10	Aplikasi di- <i>containerize</i> dengan Docker.	v	
11	Aplikasi di- <i>deploy</i> dan dapat diakses melalui internet.	v	

## DAFTAR PUSTAKA

Arhandi, P. P. 2021. "Pengembangan Sistem Informasi Perijinan Tenaga Kesehatan dengan Menggunakan Metode Backend dan Frontend." *Jurnal Teknologi Informasi*, vol. 7, p. 10.

Go Documentation. <https://go.dev/doc/>.

GeeksforGeeks. "What Is Web Scraping and How to Use It?"  
<https://www.geeksforgeeks.org/what-is-web-scraping-and-how-to-use-it/>.

HOLTE, R.C., FELNER, A., SHARON, G., STURTEVANT, N.R., and CHEN, J. 2017. "MM: A Bidirectional Search Algorithm that is Guaranteed to Meet in the Middle." *Artificial Intelligence*, 252, pp. 232–266.

Morris, S. (n.d.). "Full Stack, Front End, Back End—What Does It All Mean?"  
<https://skillcrush.com/blog/front-end-back-end-fullstack/>.

Munir, Rinaldi. 2025. "Breadth/Depth First Search (BFS/DFS) Bagian 1."  
[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf).

Munir, Rinaldi. 2025. "Breadth/Depth First Search (BFS/DFS) Bagian 2."  
[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-\(2025\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-(2025)-Bagian2.pdf).

Next.js Documentation. <https://nextjs.org/docs>.

Mozilla Developer Network (MDN). "JavaScript Guide: Language Overview."  
[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Language\\_overview](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Language_overview).

AWS Docker Documentation. <https://aws.amazon.com/docker/>.