

**LAPORAN TUGAS BESAR 2  
IF2211 STRATEGI ALGORITMA**



Dipersiapkan oleh:

Kelompok 45

Boye Mangaratua Ginting      13523127

Natalia Desiany Nursimin      13523157

Lukas Raja Agripa      13523158

**Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung  
2025**

## DAFTAR ISI

<b>DAFTAR ISI.....</b>	<b>2</b>
<b>BAB 1 DESKRIPSI TUGAS.....</b>	<b>4</b>
<b>BAB 2 LANDASAN TEORI.....</b>	<b>6</b>
2.1. Dasar Teori.....	6
2.1.1. Penjelajahan Graf.....	6
2.1.2. Algoritma Breadth First Search.....	8
2.1.3. Algoritma Depth First Search.....	10
2.1.4. Algoritma Bidirectional.....	12
2.2. Penjelasan singkat mengenai aplikasi web yang dibangun.....	14
<b>BAB 3 ANALISIS PEMECAHAN MASALAH.....</b>	<b>16</b>
3.1. Langkah-Langkah Pemecahan Masalah.....	16
3.2. Proses Pemetaan Masalah Menjadi Elemen-Elemen Algoritma DFS, BFS dan Bidirectional.....	18
3.2.1. Breadth-First Search.....	18
3.2.2. Depth-First Search.....	18
3.2.3. Bidirectional Search.....	19
3.3 Fitur Fungsional dan Arsitektur Aplikasi Web yang Dibangun.....	20
3.3.1 Fitur Fungsional.....	20
3.3.2 Arsitektur Web.....	21
Backend dikembangkan menggunakan bahasa pemrograman Go (Golang).....	21
2. Frontend (Client-Side) – React/JSX.....	21
3.4. Contoh Ilustrasi Kasus.....	22
<b>BAB 4 IMPLEMENTASI DAN PENGUJIAN.....</b>	<b>24</b>
4.1. Spesifikasi Teknis Program.....	24
4.1.1 Struktur Direktori.....	24
4.1.2 Source Code.....	25
4.1.2.1. scraper.go.....	25
4.1.2.2. search.go.....	28
4.1.2.3. recipeController.go.....	46
4.1.2.4. main.go.....	48
4.1.2.5. progress.go.....	49
4.1.2.6. response.go.....	50
4.2. Penjelasan Tata Cara Penggunaan Program.....	51
4.3. Hasil Pengujian Minimal 3 Buah Elemen.....	52
4.3.1. Uji Pencarian Elemen Brick.....	52
4.3.2. Uji Pencarian Elemen Alcohol.....	55
4.3.3. Uji Pencarian Elemen Yogurt.....	58
4.4. Analisis Hasil Pengujian.....	61
4.4.1 Analisis Pencarian Elemen Brick.....	61

4.4.2 Analisis Pencarian Elemen Alcohol.....	61
4.4.3 Analisis Pencarian Elemen Yogurt.....	61
<b>BAB 5 KESIMPULAN, SARAN, DAN REFLEKSI TENTANG TUGAS BESAR 2.....</b>	<b>62</b>
5.1. Kesimpulan.....	62
5.2. Saran.....	62
5.3. Refleksi.....	63
<b>LAMPIRAN.....</b>	<b>64</b>
Tautan Repository GitHub:.....	64
Tautan Video YouTube:.....	64
Tabel Pengecekan Fitur:.....	64
<b>DAFTAR PUSTAKA.....</b>	<b>66</b>

## BAB 1 DESKRIPSI TUGAS



Gambar 1. Little Alchemy 2  
(Sumber: <https://www.thegamer.com>)

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

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

Pada Tugas Besar 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 pemain dapat melakukan *combine* antara 2 elemen untuk menghasilkan elemen baru. Elemen turunan yang telah didapatkan dapat digunakan kembali oleh pemain untuk membentuk elemen lainnya.

## BAB 2 LANDASAN TEORI

### 2.1. Dasar Teori

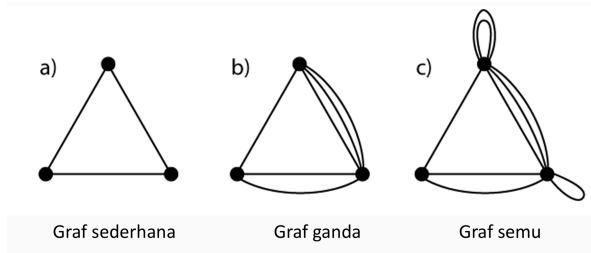
#### 2.1.1. Penjelajahan Graf

Graf merupakan salah satu struktur data paling mendasar dalam ilmu komputer yang digunakan untuk merepresentasikan hubungan antara objek. Sebuah graf umumnya digunakan untuk merepresentasikan objek-objek diskrit dan hubungan antara objek-objek tersebut. Secara formal, graf dapat didefinisikan sebagai pasangan himpunan  $G = (V, E)$ , di mana  $V$  adalah himpunan tidak-kosong dari simpul-simpul (*vertices*),  $V = \{ v_1, v_2, \dots, v_n \}$ . Sedangkan,  $E$  adalah himpunan sisi (*edges*) yang menghubungkan pasangan simpul,  $E = \{ e_1, e_2, \dots, e_n \}$ . Simpul dapat dianggap sebagai sebuah entitas, sementara sisi berfungsi untuk menggambarkan hubungan atau koneksi antar entitas tersebut. Sebuah graf tidak boleh tidak memiliki simpul, namun diperbolehkan untuk tidak mengandung sisi satupun.

Graf telah digunakan sejak lama untuk menyelesaikan berbagai permasalahan, dengan salah satu contoh awalnya adalah persoalan jembatan Königsberg pada tahun 1736 yang dipecahkan oleh Leonhard Euler. Persoalan ini berkaitan dengan apakah mungkin seseorang melalui setiap jembatan yang menghubungkan empat daratan di kota Königsberg tepat sekali dan kembali ke tempat semula. Euler berhasil memodelkan persoalan tersebut dengan merepresentasikan daratan sebagai simpul dan jembatan sebagai sisi, yang kemudian menjadi tonggak awal dari teori graf. Dalam konteks penerapannya dengan permainan Little Alchemy 2, graf dapat digunakan untuk memodelkan hubungan antar elemen, dimana setiap elemen direpresentasikan sebagai simpul, dan setiap kombinasi elemen yang menghasilkan elemen baru direpresentasikan sebagai sisi berarah dari kedua elemen pembentuk menuju elemen hasil.

Graf dapat dikelompokkan dengan ada tidaknya gelang atau sisi ganda, yaitu sebagai berikut:

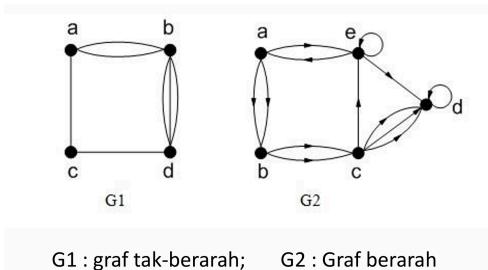
1. Graf Sederhana (*Simple Graph*): Graf tidak mengandung gelang maupun sisi ganda.
2. Graf Tak-Sederhana (*Unsimple Graph*):
  - Graf Ganda (*Multi Graph*): Graf yang mengandung sisi ganda.
  - Graf Semu (*Pseudo Graph*): Graf yang mengandung sisi gelang.



(Sumber : <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf>)

Graf berdasarkan orientasi arah pada sisinya dapat dikelompokkan menjadi dua, yaitu:

1. Graf Tak-Berarah (*Undirected Graph*): Graf yang sisinya tidak mempunyai orientasi arah.
2. Graf Berarah (*Directed Graph atau Digraph*): Graf yang setiap sisinya diberikan orientasi arah.



G1 : graf tak-berarah; G2 : Graf berarah

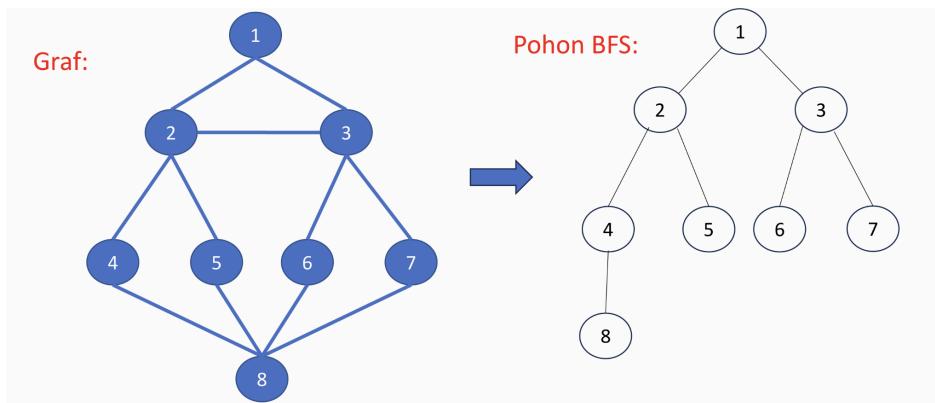
(Sumber : <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf>)

Dalam konteks tugas ini, graf yang akan digunakan adalah graf berarah karena setiap kombinasi dua elemen menghasilkan satu elemen baru yang spesifik dengan arah jelas. Sebagai contoh, kombinasi "air + earth" menghasilkan "mud", namun tidak sebaliknya.

Penjelajahan graf (*graph traversal*) merupakan proses sistematis untuk mengunjungi simpul-simpul dalam graf. Tujuan dari teknik ini adalah untuk menemukan simpul tertentu, mengevaluasi kondisi simpul, atau membentuk lintasan dari simpul awal ke simpul akhir. Dua algoritma dasar dalam penjelajahan graf adalah *Breadth First Search* (BFS) dan *Depth First Search* (DFS), yang akan dibahas lebih detail pada bagian selanjutnya. Kedua algoritma ini menjadi fondasi penting dalam implementasi pencarian recipe pada permainan Little Alchemy 2."

Proses penjelajahan graf membentuk struktur yang disebut pohon ruang status (state space tree). Dalam pohon ini, simpul merepresentasikan kondisi atau status sistem (dalam konteks Little Alchemy 2, kombinasi elemen yang telah ditemukan), cabang merepresentasikan aksi atau transisi dari satu status ke status lain (dalam konteks ini berupa kombinasi dua elemen), akar pohon adalah status awal (dalam konteks ini berupa elemen-elemen dasar), dan daun pohon merepresentasikan kondisi akhir atau solusi (dalam konteks ini berupa elemen target). Struktur pohon ini memungkinkan visualisasi dan analisis efektif dari proses pencarian.

### 2.1.2. Algoritma Breadth First Search



(Sumber: [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))

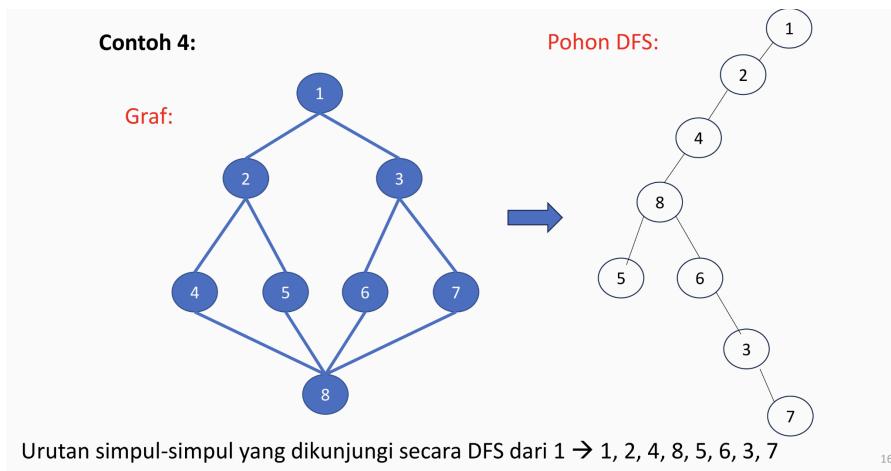
Algoritma *Breadth First Search* atau yang biasa dikenal dengan sebutan dengan sebutan BFS merupakan salah satu metode pencarian solusi dalam graf yang bersifat *uninformed* atau *blind search*, yaitu tidak menggunakan informasi tambahan dalam proses pencariannya. BFS merupakan salah satu algoritma yang dapat digunakan untuk melakukan penelusuran pada struktur data graf guna menemukan solusi dari suatu permasalahan. BFS menjelajahi graf secara sistematis dengan pendekatan melebar. Proses BFS dimulai dari simpul awal (*start node*), kemudian mengunjungi semua tetangga langsungnya terlebih dahulu sebelum berpindah ke simpul yang lebih jauh. Proses ini diulang secara bertingkat hingga seluruh simpul yang dapat dicapai telah dikunjungi atau solusi ditemukan.

Dalam pengimplementasiannya, BFS menggunakan struktur data antrian (*queue*) yang mengikuti prinsip FIFO (*First In First Out*). Ketika sebuah simpul dikunjungi, semua tetangganya yang belum pernah dikunjungi akan dimasukkan ke dalam antrian. Untuk mencegah kunjungan berulang terhadap simpul yang sama, BFS menggunakan array penanda atau tabel boolean yang mencatat apakah suatu simpul telah dikunjungi atau belum. Graf yang menjadi objek penelusuran biasanya direpresentasikan dalam bentuk matriks ketetanggaan (*adjacency matrix*) atau daftar ketetanggaan (*adjacency list*) untuk memudahkan akses terhadap simpul-simpul yang saling bertetangga.

Salah satu keunggulan dari penggunaan BFS adalah sifat *completeness*-nya, yaitu jika solusi ada, BFS akan menemukannya. Selain itu, hasil BFS juga sangat optimal. jika setiap langkah dalam graf memiliki bobot yang sama. Oleh sebab itu, BFS sangat cocok digunakan untuk menemukan *shortest path* atau jalur terpendek, seperti pada permainan puzzle, navigasi peta. Dalam konteks pencarian resep Little Alchemy 2, BFS dapat digunakan untuk menemukan kombinasi elemen yang dapat menghasilkan elemen target dengan jumlah langkah minimal. Dengan penggunaan BFS, pemain dapat memperoleh jalur penciptaan elemen secara efisien dan menghindari percobaan yang redundan atau tidak efektif.

Namun, kelemahan utama dari BFS terletak pada kebutuhan memorinya yang tinggi. Hal ini dikarenakan BFS harus menyimpan seluruh simpul pada level tertentu dalam antrian. Sehingga, kompleksitas ruangnya dapat berkembang secara eksponensial seiring dengan bertambahnya kedalaman solusi. Hal ini menjadi kendala pada graf yang sangat besar atau masalah dengan ruang status yang luas dan dalam.

### 2.1.3. Algoritma Depth First Search



16

(Sumber: [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))

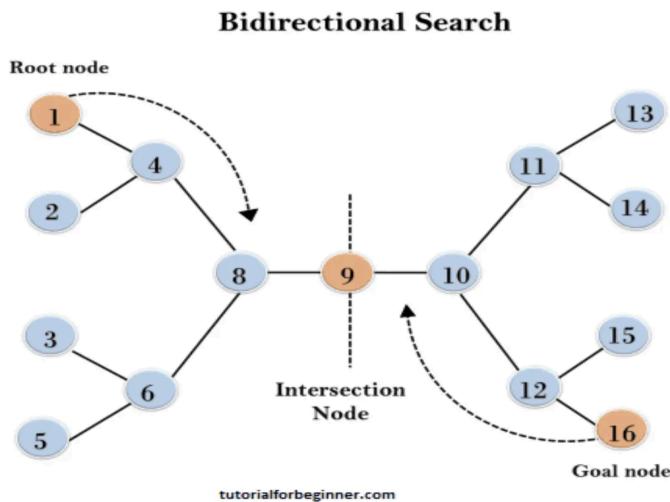
*Depth First Search* atau yang biasa dikenal dengan sebutan DFS adalah salah satu algoritma dasar dalam pencarian graf yang mengeksplorasi jalur pencarian dengan prinsip penelusuran sedalam mungkin terlebih dahulu, sebelum kembali (*backtrack*) dan menjelajahi cabang lain. Teknik ini memberikan pendekatan yang kontras dengan BFS yang menjelajah secara melebar. DFS dapat diimplementasikan secara rekursif, yaitu menggunakan fungsi yang memanggil dirinya sendiri, maupun secara iteratif dengan bantuan struktur data stack untuk melacak simpul-simpul (*nodes*) yang perlu dikunjungi berikutnya.

Secara umum, DFS bekerja dengan memilih satu jalur dan menelusurnya hingga mencapai simpul akhir atau simpul tanpa tetangga yang belum dikunjungi. Jika simpul yang dipilih tidak mengarah ke solusi, maka algoritma akan mundur (*backtrack*) ke simpul sebelumnya dan mencoba jalur alternatif lain yang belum dieksplorasi. Pola pencarian seperti ini dikenal dengan istilah *backtracking* dan menjadi salah satu ciri khas dari DFS. Proses ini berlanjut sampai seluruh simpul yang dapat dicapai telah dikunjungi, atau hingga simpul tujuan berhasil ditemukan. DFS akan terus menelusuri simpul-simpul tetangga yang belum dikunjungi hingga mencapai kedalaman maksimum, atau hingga tidak ada lagi simpul yang dapat dijelajahi dari jalur tersebut, kemudian kembali ke simpul sebelumnya untuk mencoba jalur lainnya.

DFS sangat bermanfaat dalam konteks pencarian solusi yang tidak menitikberatkan pada jalur terpendek, melainkan lebih pada eksplorasi menyeluruh terhadap semua kemungkinan jalur. Oleh karena itu, algoritma ini cocok digunakan dalam eksplorasi ruang solusi kombinatorial, seperti pada pencarian kombinasi item atau resep dalam permainan *Little Alchemy 2*. Dalam implementasinya pada konteks tersebut, simpul direpresentasikan sebagai *state* (keadaan) berupa kumpulan item yang dimiliki. Setiap *state* kemudian diproses dengan mencoba seluruh pasangan kombinasi item yang memungkinkan, menghasilkan *state* baru yang kemudian dijelajahi lebih lanjut menggunakan DFS. Proses ini dilakukan secara rekursif hingga ditemukan *state* yang

mengandung item target yang diinginkan. Untuk mencegah terjadinya perulangan eksplorasi terhadap *state* yang sama, implementasi DFS harus dilengkapi dengan struktur data penanda kunjungan, seperti *visited set* untuk menyimpan semua state yang telah dikunjungi sebelumnya.. Dengan ini, setiap *state* hanya akan diproses satu kali, meningkatkan efisiensi dan mencegah loop yang tidak diinginkan dalam proses pencarian.

#### 2.1.4. Algoritma Bidirectional



(Sumber: <https://tutorialforbeginner.com/uninformed-search-algorithms-in-ai>)

Bidirectional Search merupakan teknik pencarian yang mempercepat proses pencarian solusi dengan cara menjelajah graf dari dua arah secara bersamaan, yaitu dari simpul awal (*start node*) dan dari simpul tujuan (*goal node*). Kedua pencarian dilakukan hingga keduanya bertemu di suatu titik perpotongan. Pendekatan ini sangat bermanfaat karena secara signifikan dapat mengurangi kompleksitas ruang pencarian dibandingkan metode pencarian satu arah, seperti *Breadth First Search* (BFS) atau *Depth First Search* (DFS). Secara matematis, jika pencarian satu arah memiliki kompleksitas waktu  $O(b^d)$ , maka Bidirectional Search hanya memerlukan  $O[b^{(d/2)}]$ , dengan  $b$  sebagai branching factor dan  $d$  sebagai kedalaman solusi. Dengan demikian, metode ini sangat efisien dikarenakan dapat membuat jumlah simpul yang harus dijelajahi hanya sekitar akar kuadrat dari jumlah total simpul pada pencarian satu arah.

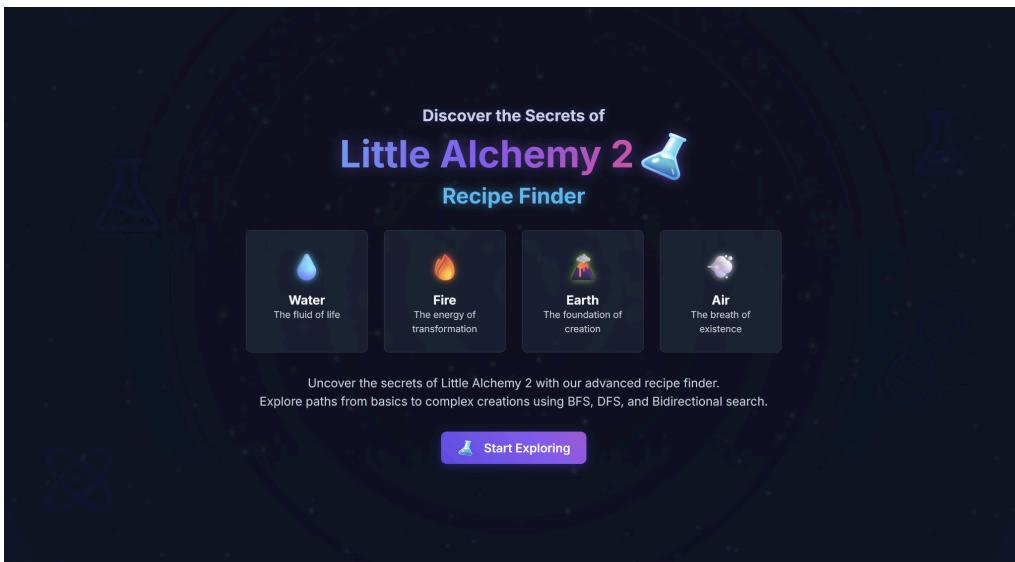
Dalam konteks Tugas Besar 2 IF2211 Strategi Algoritma yang membahas eksplorasi kombinasi resep dalam permainan *Little Alchemy 2*, algoritma *Bidirectional Search* dapat diterapkan untuk mempercepat pencarian kombinasi item menuju target item tertentu. Implementasi algoritma ini melibatkan dua proses pencarian, yaitu pencarian maju (*forward search*) dari *initial state* yang berisi kumpulan item dasar yang tersedia, dan pencarian mundur (*backward search*) dari *goal state*, yaitu state yang mengandung item target. Pada pencarian maju, algoritma mencoba seluruh kombinasi dua item yang valid sesuai dengan daftar resep. Sebaliknya, pada pencarian mundur, algoritma mencoba mendekomposisi item target menjadi pasangan-pasangan item penyusunnya, berdasarkan data resep yang dibalik (*reverse recipe*).

Kedua arah pencarian ini masing-masing menggunakan *visited state* untuk melacak state yang telah dikunjungi, serta *frontier* (antrian state yang akan dikembangkan). Proses ini hanya akan dihentikan ketika ditemukan *state* yang sama dalam kedua arah pencarian, yaitu sebuah titik

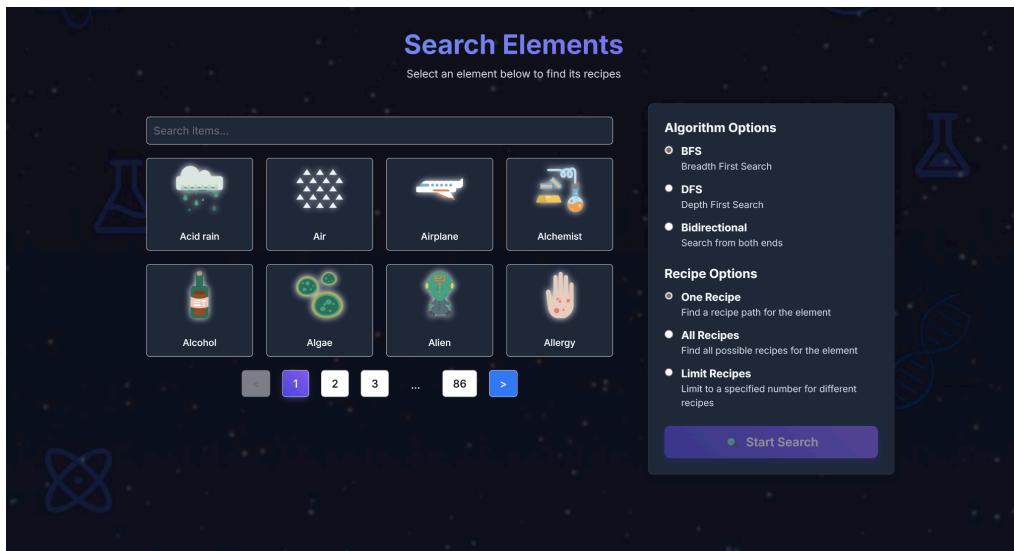
temu. Setelah titik temu ditemukan, solusi lengkap dapat dirangkai dengan menggabungkan jalur dari *initial state* ke titik temu dan dari titik temu ke *goal state*.

Kelebihan utama dari pendekatan *Bidirectional Search* dalam konteks permainan *Little Alchemy 2* terletak pada kemampuannya untuk mengurangi jumlah kombinasi yang perlu dicoba, terutama ketika banyak langkah dibutuhkan untuk mencapai item target. Namun, tantangan terbesar dalam penerapannya adalah efisiensi pencarian mundur, karena tidak semua item dalam permainan memiliki informasi eksplisit tentang pasangan penyusunnya. Untuk mengatasi hal ini, diperlukan struktur data pendukung seperti *reverse recipe map* yang dapat memetakan setiap item ke satu atau lebih pasangan komponen pembentuknya, serta manajemen struktur *data visited* dan *frontier* yang baik untuk mencegah eksplorasi *state* yang berulang-ulang.

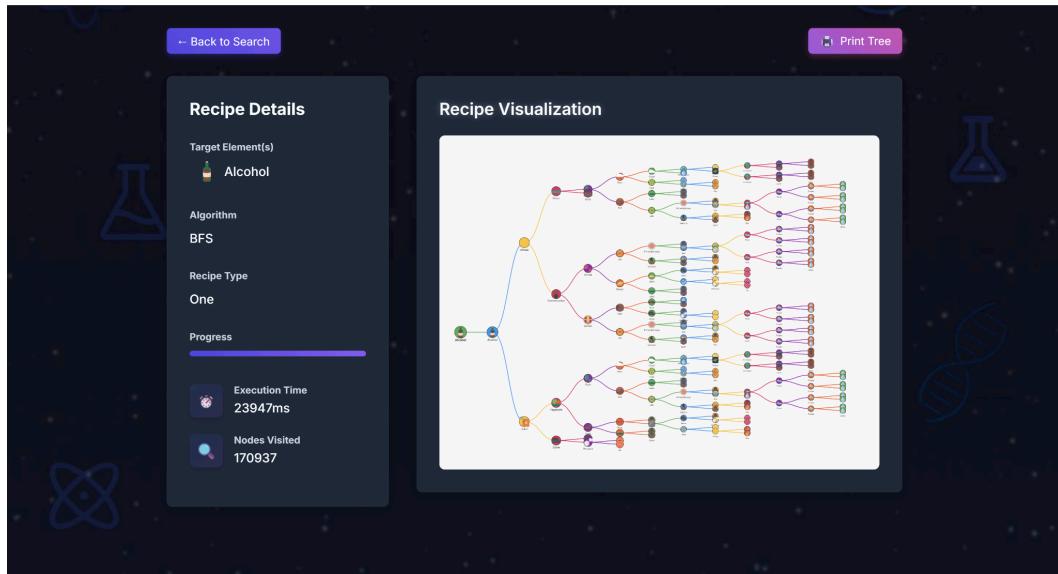
## 2.2. Penjelasan singkat mengenai aplikasi web yang dibangun



Gambar 2.2.1 Tampilan awal pada website



Gambar 2.2.2 Tampilan pada web untuk memilih elemen dan jenis algoritma



**Gambar 2.2.3 Contoh tampilan detail *recipe* dan visualisasi *tree* pada web**

Aplikasi web yang dibangun berjudul “*Discover the Secrets of Little Alchemy 2 Recipe Finder*” Aplikasi web ini dibangun untuk membantu pencarian *recipe* untuk permainan *Little Alchemy 2*. Tujuan utama aplikasi ini adalah membantu pengguna dalam menemukan kombinasi elemen yang tepat untuk membentuk elemen target secara cepat, akurat, dan efisien. Pada aplikasi web ini, melalui antarmuka yang interaktif, pengguna dapat memilih elemen target yang mereka inginkan menggunakan fitur pencarian yang tersedia. Selain itu, pengguna juga dapat memilih jenis algoritma pencarian yang mereka ingin gunakan, seperti *Breadth First Search* (BFS) , *Depth First Search* (DFS), atau *Bidirectional Search*, sesuai dengan kebutuhan mereka.. Pengguna juga diberikan opsi untuk memilih tipe *recipe* yang ingin ditampilkan, seperti *one recipe*, *all recipes*, maupun *limit recipes*.

Setelah proses pencarian selesai dilakukan, web akan menampilkan informasi detail mengenai proses pencarian yang meliputi waktu eksekusi dan jumlah simpul yang dikunjungi.. Hasil *recipe* kemudian akan divisualisasikan dalam bentuk *tree*, yang menunjukkan langkah-langkah kombinasi elemen dari elemen dasar hingga terbentuknya elemen target. Visualisasi ini dibuat dengan tujuan untuk memudahkan pengguna dalam memahami jalur pembentukan elemen dengan lebih jelas dan interaktif.

Aplikasi web ini dikembangkan dengan menggunakan React.js pada bagian *frontend* untuk membangun antarmuka pengguna, serta Golang pada bagian *backend* untuk menangani logika algoritma pencarian dan pengolahan data. dengan pendekatan arsitektur ini, aplikasi mampu memberikan pengalaman yang optimal bagi pengguna, baik dari sisi tampilan yang interaktif maupun dari segi performa.

## BAB 3 ANALISIS PEMECAHAN MASALAH

### 3.1. Langkah-Langkah Pemecahan Masalah

Permasalahan utama yang harus diselesaikan dalam Tugas Besar 2 Mata Kuliah IF2211 Strategi Algoritma ini adalah untuk menemukan kombinasi elemen-elemen dalam permainan *Little Alchemy 2* yang dapat membentuk elemen target tertentu. Permainan ini dibuat berbasis eksplorasi kombinasi, di mana pemain dapat menggabungkan dua elemen untuk menciptakan elemen baru. Proses ini bersifat rekursif dan iteratif, yaitu elemen yang telah terbentuk bisa digabungkan kembali untuk menciptakan elemen lainnya. Tantangan utamanya adalah untuk menelusuri ruang kombinasi yang besar ini dengan efisien untuk mencapai elemen target.

Untuk menyelesaikan permasalahan ini, langkah-langkah yang dilakukan adalah sebagai berikut:

#### 1. Menentukan Elemen Target dan Menyiapkan Struktur Data

Langkah pertama dimulai dengan pengguna harus memilih elemen target yang ingin dicari. Elemen ini akan menjadi tujuan akhir dari pencarian jalur kombinasi dalam pencarian resep permainan *Little Alchemy 2*. Sistem telah memiliki data kombinasi elemen-elemen yang tersedia dalam bentuk struktur data di *database*, di mana setiap elemen dapat dibentuk dari kombinasi dua elemen lainnya. Elemen target ini kemudian ditetapkan sebagai *goal node* dalam proses pencarian jalur.

#### 2. Pemilihan Algoritma dan Tipe Resep oleh Pengguna

Pada langkah selanjutnya, pengguna diharuskan untuk memilih tipe algoritma pencarian yang akan digunakan dalam proses pembentukan elemen, yang terdiri dari tiga pilihan utama, yaitu dengan *Breadth First-Search*, *Depth First-Search*, serta *Bidirectional Search*. Setelah itu, pengguna juga dapat menentukan tipe hasil yang diinginkan, yaitu *one recipe*, *all recipes* dan *limit recipes*. Pada *one recipe*, hanya akan ditampilkan satu jenis resep. Pada *all recipes*, akan ditampilkan semua kombinasi yang mungkin. Terakhir, untuk *limit recipes* akan diberikan jumlah kombinasi yang sesuai dengan jumlah input pengguna.

#### 3. Pemodelan Permasalahan sebagai Graf Pencarian

Setelah, pengguna telah menampilkan semua spesifikasi yang diinginkan. Maka, *backend* akan memodelkan seluruh proses pencarian kombinasi elemen sebagai sebuah graf. Pada graf ini, setiap simpul mewakili sebuah elemen, sedangkan setiap sisi merepresentasikan kombinasi dua elemen yang dapat membentuk elemen baru. Proses ini menghasilkan graf dinamis yang akan dapat ditelusuri menggunakan algoritma pencarian yang telah dipilih oleh pengguna.

#### **4. Proses Pencarian Jalur Kombinasi oleh Backend**

Selanjutnya, *backend* akan menjalankan algoritma pencarian yang dipilih (BFS, DFS, atau Bidirectional) untuk mencari kombinasi elemen dari elemen dasar hingga membentuk elemen target. Selama proses ini, sistem juga mencatat metrik penting seperti jumlah simpul yang dikunjungi dan waktu eksekusi. Jika ditemukan satu atau lebih jalur kombinasi yang valid, hasilnya akan disimpan dalam struktur data yang siap dikirim ke frontend.

#### **5. Pengiriman dan Penampilan Hasil ke Frontend**

Seluruh hasil pencarian yang diperoleh dari *backend* akan dikirim ke *frontend* dalam bentuk struktur data yang telah diproses. Hal ini mencakup jalur kombinasi elemen yang diperoleh, informasi mengenai lama waktu eksekusi dalam bentuk ms, serta jumlah simpul yang dikunjungi pada pencarian. Bagian *frontend* kemudian akan menampilkan hasil ini kepada pengguna dalam bentuk visualisasi *tree* yang akan menunjukkan urutan elemen dasar hingga terbentuknya elemen target. Lalu, seluruh keterangan akan dimunculkan pada bagian *recipe details*. Hal ini mencakup *target element* yang telah dipilih pengguna, jenis tipe algoritma yang dipilih pengguna, jenis tipe resep yang dipilih, lama waktu eksekusi, serta jumlah node yang dikunjungi.

## 3.2. Proses Pemetaan Masalah Menjadi Elemen-Elemen Algoritma DFS, BFS dan Bidirectional

Dalam aplikasi web ini, setiap elemen dalam permainan *Little Alchemy 2* direpresentasikan sebagai simpul (*node*) pada sebuah graf terarah. Graf ini digunakan untuk memodelkan hubungan antar elemen berdasarkan kombinasi resep yang memungkinkan pembentukan elemen baru dari elemen-elemen dasar. Setiap *node* yang ada memiliki atribut boolean bernama `cekValid`. Atribut ini berfungsi sebagai penanda apakah elemen tersebut telah berhasil ditemukan melalui proses pencarian resep. Jika node elemen yang dicari sudah mendapatkan nilai *True* pada atribut `cekValid`, maka recipe sudah ditemukan. Hal ini mempermudah proses validasi hasil pencarian dalam algoritma *Breadth-First Search* (BFS), *Depth-First Search* (DFS), dan *Bidirectional Search*.

### 3.2.1. Breadth-First Search

Algoritma BFS pada program ini diterapkan dengan pendekatan *traversal level-order* menggunakan struktur data *Queue*. Pendekatan ini digunakan untuk menjamin bahwa simpul akan diperiksa berdasarkan urutan kedalaman dari akar ke daun.

1. Sebuah *Queue* akan dibentuk untuk menentukan prioritas urutan pengecekan node.
2. Node mula-mula merupakan elemen yang dicari, sesuai dengan inputan pengguna.
3. Node ini akan dicek, apakah terdapat recipe yang bisa membentuk node tersebut. Jika ada, akan dibentuk node-node baru yang dapat membentuk node tersebut. Node yang sedang dicek akan dihapus antriannya dari *Queue*, sedangkan node-node baru akan ditambahkan ke *Queue* (LIFO), yakni antrian pengecekan urutan agar sesuai dengan algoritma *Breadth-First Search* (BFS).
4. Jika tidak ada pembentuknya (Null + Null), atau kedua *node* yang menjadi pembentuknya sudah bernilai *True* pada atribut `cekValid` pada atribut booleannya, maka *node* tersebut otomatis memperoleh *True* pada atribut `cekValid`-nya juga.
5. Proses 3 dan 4 akan terus berulang hingga *node* elemen yang dicari (node paling awal) mendapatkan nilai *True* pada bagian `cekValid`.
6. Hasil akhir berbentuk sebuah *Tree* akan dikembalikan dan nantinya akan ditampilkan pada bagian *TreeVisualizer*.

### 3.2.2. Depth-First Search

Algoritma DFS pada program ini menggunakan pendekatan traversal mendalam terlebih dahulu (*depth-first*), yang diimplementasikan dengan memanfaatkan struktur data *Stack* (tumpukan).

1. Sebuah *Stack* akan dibentuk untuk menentukan prioritas urutan pengecekan *node*.
2. *Node* awal merupakan elemen yang dicari, sesuai dengan inputan pengguna.

3. *Node* ini akan dicek, apakah terdapat *recipe* yang bisa membentuk node tersebut. Jika *Node* ini akan dicek, apakah memiliki *recipe* yang dapat membentuknya. Jika ada, *node-node* baru yang menjadi pembentuk akan dibentuk. *Node* yang sedang diperiksa dihapus dari *Stack*, sementara *node* pembentuk ditambahkan ke *Stack* dengan mengikuti urutan LIFO agar sesuai dengan karakteristik algoritma *Depth-First Search*.
4. Jika tidak ada pembentuknya (Null + Null), atau kedua *node* yang menjadi pembentuknya sudah bernilai *True* pada atribut *cekValid*, maka *node* tersebut otomatis memperoleh *True* pada atribut *cekValid*-nya juga.
5. Proses 3 dan 4 akan terus berulang hingga node elemen yang dicari (*node* paling awal) mendapatkan nilai *True* pada bagian *cekValid*.
6. Hasil akhir berupa sebuah *Tree Recipe* akan dikembalikan yang nantinya akan ditampilkan pada bagian *TreeVisualizer*.

### 3.2.3. Bidirectional Search

Metode ini memiliki dasar yang serupa dengan BFS, tetapi perbedaannya terletak pada simpul awal pencarian yang tidak hanya berupa elemen target, melainkan juga elemen-elemen dasar.

1. Beberapa *Queue* akan dibentuk untuk menentukan prioritas urutan pengecekan *node*, dimana satu *Queue* digunakan untuk pencarian BFS dari elemen yang dicari, dan sisanya untuk pencarian BFS dari elemen dasar.
2. Pencarian dilakukan secara dua arah, dimana satu pencarian mengarah ke elemen-elemen dasar dari elemen yang dicari, sedangkan arah lainnya ialah dimulai dari elemen-elemen dasar ke arah elemen-elemen yang lebih kompleks.
3. Singkatnya, terdapat lebih dari satu alur pencarian BFS pada algoritma ini.
4. Sama seperti alur BFS, namun proses pencarian pada program akan berhenti saat elemen *node* pada pencarian kedua arah sudah sama. Saat dua *node* identik ditemukan dari arah yang berbeda, nilai *True* diberikan pada atribut *cekValid* milik *node* tersebut, menandakan bahwa jalur *recipe* telah ditemukan.
5. Hasil akhir berupa *Tree Recipe* akan dikembalikan dan nantinya ditampilkan pada bagian *TreeVisualizer*.

### **3.3 Fitur Fungsional dan Arsitektur Aplikasi Web yang Dibangun**

#### **3.3.1 Fitur Fungsional**

Aplikasi web ini dikembangkan untuk membantu pengguna mencari kombinasi (*recipe*) dalam permainan *Little Alchemy 2* menggunakan algoritma pencarian. Fitur-fitur fungsional yang tersedia antara lain:

##### **1. Pencarian Recipe Berdasarkan Nama Item Tujuan**

Pengguna dapat memasukkan nama elemen yang ingin dicari dan sistem akan mencarikan recipe untuk membuat elemen tersebut.

##### **2. Pilihan Algoritma Pencarian**

Pengguna dapat memilih salah satu dari tiga algoritma pencarian yang tersedia:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Bidirectional Search

##### **3. Jenis Pencarian Recipe**

Pengguna dapat memilih salah satu jenis pencarian:

- **One Recipe:** Menampilkan satu solusi untuk membuat elemen.
- **All Recipes:** Menampilkan semua solusi yang memungkinkan.
- **Limit Recipes:** Menampilkan sejumlah solusi berdasarkan batas (limit) yang ditentukan oleh pengguna.

##### **4. Tampilan Interaktif**

- Antarmuka pengguna sederhana dan intuitif untuk memilih elemen target, jenis algoritma, dan tipe pencarian.
- Hasil ditampilkan dalam bentuk langkah-langkah pembuatan elemen (dari kombinasi dasar hingga target).

#### **3.3.2 Arsitektur Web**

Aplikasi web pencarian recipe *Little Alchemy 2* ini dibangun dengan pendekatan arsitektur client-server, yang terdiri dari dua komponen utama: backend server (Golang) dan

frontend (JSX/React). Arsitektur ini memisahkan tanggung jawab antara logika pemrosesan dan antarmuka pengguna untuk menjaga modularitas dan skalabilitas aplikasi.

## 1. Backend (Server-Side) – Golang

Backend dikembangkan menggunakan bahasa pemrograman **Go (Golang)**.

Berfungsi untuk:

- Menyediakan **REST API** yang menangani permintaan dari frontend.
- Menjalankan **algoritma pencarian** recipe (BFS, DFS, Bidirectional Search).
- Mengelola request pencarian berdasarkan mode:
  - One Recipe
  - All Recipes
  - Limit Recipe (dengan input batasan integer)
- Melakukan validasi input dan mengembalikan hasil pencarian dalam format yang bisa dibaca oleh Visualizer pada front end.

## 2. Frontend (Client-Side) – React/JSX

Antarmuka pengguna dibangun dengan **React** menggunakan **JSX**.

Bertugas untuk:

- Menyediakan tampilan yang interaktif bagi pengguna untuk memilih metode pencarian dan jenis pencarian.
- Mengirim permintaan ke backend.
- Menampilkan hasil pencarian dalam bentuk tree yang dapat di zoom-in dan zoom-out, serta paginasi jika terdapat lebih dari satu resep yang diperoleh.

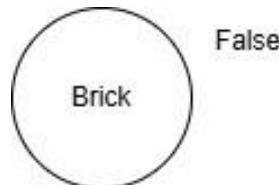
### 3.4. Contoh Ilustrasi Kasus

Pengguna ingin mencari seluruh resep dari elemen “Brick” dengan metode BFS - One Recipe.

Backend menerima permintaan dari pengguna dan memproses pencarian. Alurnya ialah sebagai berikut:

### Iterasi 1

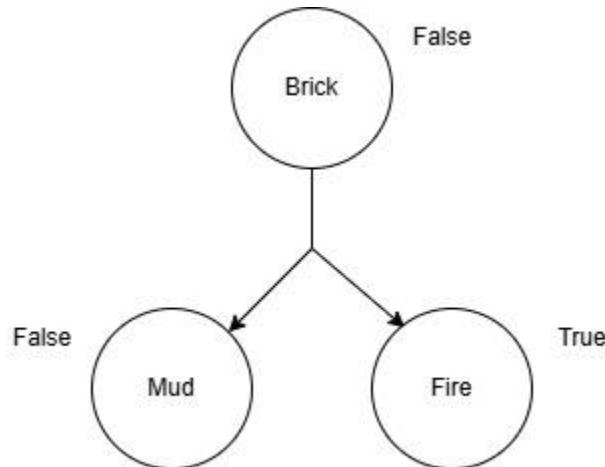
Node pencarian hanyalah “Brick” itu sendiri. cekValid dari Brick belum bernilai True karena ia memiliki resep penyusun, dan kedua node di bawahnya belum keduanya memiliki cekValid yang bernilai True.



Gambar 3.4.1 Ilustrasi Iterasi 1

### Iterasi 2

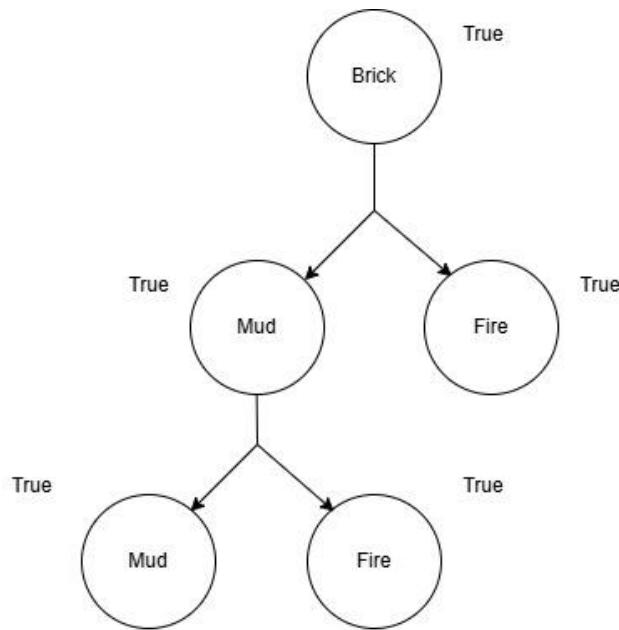
Karena hanya terdapat recipe  $\text{Brick} = \text{Mud} + \text{Fire}$  yang valid, maka node bertambah 2 menjadi seperti ilustrasi di bawah ini. Fire memiliki nilai cekValid = True sedangkan Mud tidak.



Gambar 3.4.2 Ilustrasi Iterasi 2

### Iterasi 3

Pencarian berlanjut ke elemen “Mud”, dan menghasilkan dua node baru dimana  $\text{Mud} = \text{Earth} + \text{Fire}$ . Karena node pembentuk Mud keduanya bernilai True pada cekValid, maka Mud bernilai True pada cekValidnya. Karena Brick memiliki Mud dan Fire bernilai True, maka Brick menjadi bernilai True juga pada cekValid. Saat elemen yang dicari bernilai True, maka pencarian one recipe selesai.



Gambar 3.4.3 Ilustrasi Iterasi 3

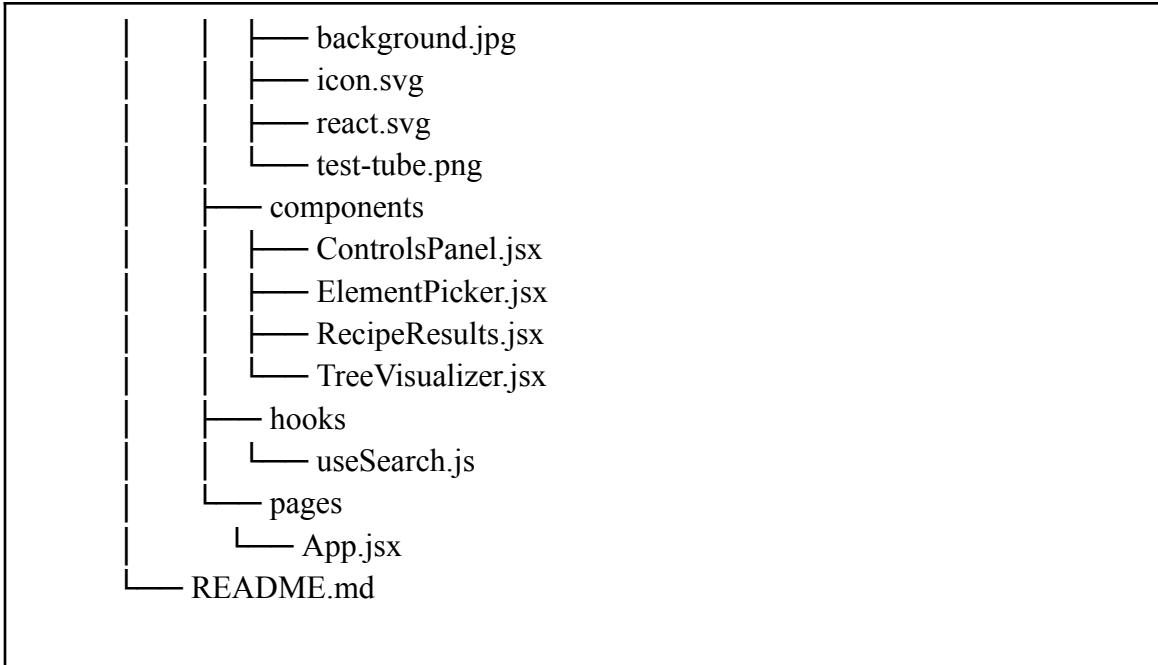
## BAB 4 IMPLEMENTASI DAN PENGUJIAN

### 4.1. Spesifikasi Teknis Program

#### 4.1.1 Struktur Direktori

Program yang kami buat memiliki struktur direktori sebagai berikut:

```
Tubes2_masih-kuat-kok-Taylor-s-version (Root Directory)
├── eslint.config.js
├── index.html
├── package-lock.json
├── package.json
├── vite.config.js
└── backend
    ├── Dockerfile
    ├── go.mod
    ├── go.sum
    ├── main.go
    ├── controllers
    │   └── recipeController.go
    ├── services
    │   └── search.go
    └── utils
        └── response.go
    └── database
        ├── alchemy.db
        ├── buat-mapper.js
        └── mapper2.json
    └── doc
        └── masih kuat kok!(Taylor's version)
    └── frontend
        ├── Dockerfile
        ├── public
        │   ├── App.css
        │   ├── index.css
        │   └── vite.svg
        └── src
            ├── main.jsx
            ├── assets
            └── background.css
```



#### 4.1.2 Source Code

##### 4.1.2.1. scraper.go

```

package main

import (
    "encoding/csv"
    "fmt"
    "log"
    "os"
    "strings"

    "database/sql"

    "github.com/PuerkitoBio/goquery"
    "github.com/mattn/go-sqlite3" // Untuk SQLite
)

func main() {
    // Buka file HTML lokal
    f, err := os.Open("Elements (Little Alchemy 2) _ Little Alchemy Wiki _ Fandom.html")
    if err != nil {
        log.Fatalf("Gagal membuka file HTML: %v", err)
    }
}

```

```

    }

    defer f.Close()

    // Buat dokument dari file reader
    doc, err := goquery.NewDocumentFromReader(f)
    if err != nil {
        log.Fatalf("Gagal membuat dokumen dari file: %v", err)
    }

    // Buat file CSV untuk gabungan tabel
    file, err := os.Create("alchemy.csv")
    if err != nil {
        log.Printf("Gagal membuat file alchemy.csv: %v", err)
        return
    }
    defer file.Close()

    writer := csv.NewWriter(file)
    defer writer.Flush()

    // Tulis header CSV
    writer.Write([]string{"Element", "Item1", "Item2"})

    // Setup SQLite database
    db, err := sql.Open("sqlite3", "./alchemy.db")
    if err != nil {
        log.Fatalf("Gagal membuka SQLite database: %v", err)
    }
    defer db.Close()

    // Buat tabel dalam database jika belum ada
    _, err = db.Exec(`CREATE TABLE IF NOT EXISTS elements (
        element TEXT,
        item1 TEXT,
        item2 TEXT
    );
`)
    if err != nil {
        log.Fatalf("Gagal membuat tabel: %v", err)
    }

    // Menyimpan data ke database dan CSV
    doc.Find("h3").Each(func(i int, s *goquery.Selection) {

```

```

headline := s.Find(".mw-headline")
sectionID, exists := headline.Attr("id")
if !exists {
    return
}

table := s.NextAllFiltered("table").First()
if table.Length() == 0 {
    return
}

// Memasukkan data ke dalam SQLite dan CSV
table.Find("tr").Each(func(i int, tr *goquery.Selection) {
    if i == 0 {
        return // skip header
    }
    tds := tr.Find("td")
    if tds.Length() < 2 {
        return
    }

    element := strings.TrimSpace(tds.Eq(0).Text())

    liFound := false
    tds.Eq(1).Find("li").Each(func(_ int, li *goquery.Selection) {
        text := strings.TrimSpace(li.Text())
        if strings.Contains(text, "+") {
            // Pisahkan berdasarkan '+'
            parts := strings.Split(text, "+")
            if len(parts) != 2 {
                return
            }
            item1 := strings.TrimSpace(parts[0])
            item2 := strings.TrimSpace(parts[1])

            // Menulis ke file CSV
            writer.Write([]string{element, item1, item2})

            // Menulis ke SQLite
            insertDataToSQLite(db, &element, &item1, &item2)
        }
        liFound = true
    })
})

```

```

        if !liFound {
            // Tidak ada '+' dalam <li>, tulis satu baris NULL ke CSV dan
            // SQLite
            writer.Write([]string{element, "", ""})
            insertDataToSQLite(db, &element, nil, nil)
        }
    })

    fmt.Println("Saved:", sectionID)
}

}

// Fungsi untuk memasukkan data ke SQLite
func insertDataToSQLite(db *sql.DB, element, item1, item2 *string) {

    // Menyisipkan data ke dalam database SQLite
    _, err := db.Exec("INSERT INTO elements (element, item1, item2) VALUES
    (?, ?, ?)", element, item1, item2)
    if err != nil {
        log.Printf("Gagal memasukkan data ke SQLite: %v", err)
    }
}

```

#### 4.1.2.2. search.go

```

// Complete search.go implementation with full decomposition to basic
elements
package services

import (
    "database/sql"
    "encoding/json"
    "log"
    "os"
    "time"

    _ "github.com/mattn/go-sqlite3"
)

var db *sql.DB
var mapper map[string]string

```

```

func init() {
    var err error
    db, err = sql.Open("sqlite3", "../database/alchemy.db")
    if err != nil {
        log.Fatalf("Gagal membuka database: %v", err)
    } else {
        log.Printf("Database ditemukan")
    }

    file, err := os.Open("../database/mapper2.json")
    if err != nil {
        log.Fatalf("Gagal membuka mapper.json: %v", err)
    }
    defer file.Close()
    if err := json.NewDecoder(file).Decode(&mapper); err != nil {
        log.Fatalf("Gagal mendekode mapper.json: %v", err)
    }
}

type Node struct {
    Name      string
    Children [] *Node
}

type RecipeStep struct {
    Result string
    Item1  string
    Item2  string
}

// Get all basic elements (Water, Fire, Earth, Air, etc.)
func getBasicElements() []string {
    basicElements := []string{}
    rows, err := db.Query("SELECT DISTINCT element FROM elements WHERE
element NOT IN (SELECT DISTINCT element FROM elements WHERE item1 IS NOT NULL
AND item2 IS NOT NULL)")
    if err == nil {
        defer rows.Close()
        for rows.Next() {
            var element string
            if err := rows.Scan(&element); err == nil {
                basicElements = append(basicElements, element)
            }
        }
    }
}

```

```

        }
    }
    return basicElements
}

// Check if an element is a basic element
func isBasicElement(element string, basicElements []string) bool {
    for _, basic := range basicElements {
        if element == basic {
            return true
        }
    }
    return false
}

// Helper function to get all direct combinations that create an element
func getDirectCombinations(elementName string) ([]struct {
    Item1, Item2 string
}, error) {
    var combinations []struct {
        Item1, Item2 string
    }

    rows, err := db.Query("SELECT item1, item2 FROM elements WHERE element = ? AND item1 IS NOT NULL AND item2 IS NOT NULL", elementName)
    if err != nil {
        return combinations, err
    }
    defer rows.Close()

    for rows.Next() {
        var item1, item2 string
        if err := rows.Scan(&item1, &item2); err != nil {
            continue
        }
        combinations = append(combinations, struct {
            Item1, Item2 string
        }{Item1: item1, Item2: item2})
    }

    return combinations, nil
}

// Helper function for default result when no recipe is found
}

```

```

func getDefaultResult(elementName string) []interface{} {
    return []interface{}{
        map[string]interface{}{
            "name":      elementName,
            "image":     mapper[elementName],
            "children":  []interface{}{},
            "recipe":    []string{"This is a basic element or no recipe found"},
        },
    }
}

// Format recipe steps for display
func formatRecipeSteps(steps []RecipeStep) []string {
    if len(steps) == 0 {
        return []string{}
    }

    formattedSteps := make([]string, 0, len(steps))
    for _, step := range steps {
        formattedSteps = append(formattedSteps, step.Result+" =
"+step.Item1+" + "+step.Item2)
    }
    return formattedSteps
}

//=====
// BFS IMPLEMENTATION
//=====

// BFS for recipe search
func BFS(elementName string, recipeType string, maxRecipes int) ([]interface{}, int, float64) {
    start := time.Now()
    nodesVisited := 0

    // Check if element exists in database
    var exists bool
    err := db.QueryRow("SELECT EXISTS(SELECT 1 FROM elements WHERE element = ?)", elementName).Scan(&exists)
    if err != nil || !exists {
        return getDefaultResult(elementName), nodesVisited,
float64(time.Since(start).Milliseconds())
    }
}

```

```

    // Get all basic elements
    basicElements := getBasicElements()

    // Check if this is already a basic element
    if isBasicElement(elementName, basicElements) {
        return getDefaultResult(elementName), nodesVisited,
float64(time.Since(start).Milliseconds())
    }

    // Find all complete recipes using BFS
    allRecipes := findAllRecipesBFS(elementName, basicElements,
&nodesVisited)

    // If no recipes found, return default
    if len(allRecipes) == 0 {
        return getDefaultResult(elementName), nodesVisited,
float64(time.Since(start).Milliseconds())
    }

    // Limit recipes based on recipeType
    var recipesToUse [][]RecipeStep
    if recipeType == "One" && len(allRecipes) > 0 {
        recipesToUse = allRecipes[:1]
    } else if recipeType == "Limit" && len(allRecipes) > maxRecipes {
        recipesToUse = allRecipes[:maxRecipes]
    } else {
        recipesToUse = allRecipes
    }

    // Convert recipes to result format
    var results []interface{}
    for _, recipe := range recipesToUse {
        // Create tree representation
        treeRoot := createRecipeTree(elementName, recipe)
        results = append(results, treeRoot)
    }

    return results, nodesVisited, float64(time.Since(start).Milliseconds())
}

// Function to find all possible recipes for an element using BFS
func findAllRecipesBFS(elementName string, basicElements []string,
nodesVisited *int) [][]RecipeStep {

```

```

var allRecipes [][]RecipeStep

// Queue for BFS
type QueueItem struct {
    Element string
    Path    []RecipeStep
    Explored map[string]bool // Track which elements are already explored
in this path
}

// Start with the target element
queue := []QueueItem{{
    Element: elementName,
    Path:    []RecipeStep{},
    Explored: make(map[string]bool),
}}
}

// Keep track of combinations we've added
processedCombinations := make(map[string]bool)

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

    // Skip if we've already explored this element in the current path to
    // avoid cycles
    if current.Explored[current.Element] {
        continue
    }

    // Mark this element as explored in this path
    explored := make(map[string]bool)
    for k, v := range current.Explored {
        explored[k] = v
    }
    explored[current.Element] = true

    // Check if we've reached all basic elements
    allBasic := true
    for _, step := range current.Path {
        // Check if both ingredients are basic
        item1Basic := isBasicElement(step.Item1, basicElements)
        item2Basic := isBasicElement(step.Item2, basicElements)
    }
}

```

```

        if !item1Basic || !item2Basic {
            allBasic = false
            break
        }
    }

    // If all elements in the path are decomposed to basic elements and
    // we have steps
    if allBasic && len(current.Path) > 0 {
        // Create a unique key for this recipe
        recipeKey := ""
        for _, step := range current.Path {
            recipeKey += step.Result + step.Item1 + step.Item2 + "|"
        }

        // Only add if we haven't processed this exact recipe before
        if !processedCombinations[recipeKey] {
            processedCombinations[recipeKey] = true
            allRecipes = append(allRecipes, current.Path)
        }
        continue
    }

    // Get all combinations for this element
    combinations, err := getDirectCombinations(current.Element)
    if err != nil || len(combinations) == 0 {
        // If there are no combinations (basic element or missing), and
        // this is the target element
        if current.Element == elementName {
            // Try next element in queue
            continue
        }

        // This path can't be completed, don't add to results
        continue
    }

    // Process each combination
    for _, combo := range combinations {
        // Create new step
        newStep := RecipeStep{
            Result: current.Element,
            Item1:  combo.Item1,

```

```

        Item2: combo.Item2,
    }

    // Create new path with this step
    newPath := make([]RecipeStep, len(current.Path))
    copy(newPath, current.Path)
    newPath = append(newPath, newStep)

    // Add both ingredients to queue to continue exploration
    if !isBasicElement(combo.Item1, basicElements) {
        queue = append(queue, QueueItem{
            Element: combo.Item1,
            Path:     newPath,
            Explored: explored,
        })
    }

    if !isBasicElement(combo.Item2, basicElements) {
        queue = append(queue, QueueItem{
            Element: combo.Item2,
            Path:     newPath,
            Explored: explored,
        })
    }

    // If both ingredients are basic elements, check if we have a
    complete path
    if isBasicElement(combo.Item1, basicElements) &&
    isBasicElement(combo.Item2, basicElements) {
        // Create a unique key for this recipe
        recipeKey := ""
        for _, step := range newPath {
            recipeKey += step.Result + step.Item1 + step.Item2 + "|"
        }

        // Only add if we haven't processed this exact recipe before
        if !processedCombinations[recipeKey] {
            processedCombinations[recipeKey] = true
            allRecipes = append(allRecipes, newPath)
        }
    }
}

```

```

        return allRecipes
    }

    // Create a tree representation for a recipe
    func createRecipeTree(elementName string, recipe []RecipeStep)
    map[string]interface{} {
        return map[string]interface{}{
            "name":      elementName,
            "image":     mapper[elementName],
            "children":  buildElementTree(elementName, recipe),
            "recipe":    formatRecipeSteps(recipe),
        }
    }

    // Build tree for an element recursively
    func buildElementTree(elementName string, recipe []RecipeStep)
    []map[string]interface{} {
        // Find the step for this element
        var stepForElement *RecipeStep
        for i, step := range recipe {
            if step.Result == elementName {
                stepForElement = &recipe[i]
                break
            }
        }

        // If not found, this is a basic element
        if stepForElement == nil {
            return []map[string]interface{}{}
        }

        // Create nodes for ingredients
        item1Node := map[string]interface{}{
            "name":  stepForElement.Item1,
            "image": mapper[stepForElement.Item1],
        }

        item2Node := map[string]interface{}{
            "name":  stepForElement.Item2,
            "image": mapper[stepForElement.Item2],
        }

        // Recursively build trees for ingredients
        item1Node["children"] = buildElementTree(stepForElement.Item1, recipe)
    }
}

```

```

        item2Node["children"] = buildElementTree(stepForElement.Item2, recipe)

        return []map[string]interface{}{item1Node, item2Node}
    }

//=====
// DFS IMPLEMENTATION
//=====

// DFS for recipe search
func DFS(elementName string, recipeType string, maxRecipes int)
([]interface{}, int, float64) {
    start := time.Now()
    nodesVisited := 0

    // Check if element exists in database
    var exists bool
    err := db.QueryRow("SELECT EXISTS(SELECT 1 FROM elements WHERE element = ?)", elementName).Scan(&exists)
    if err != nil || !exists {
        return getDefaultResult(elementName), nodesVisited,
float64(time.Since(start).Milliseconds())
    }

    // Get all basic elements
    basicElements := getBasicElements()

    // Check if this is already a basic element
    if isBasicElement(elementName, basicElements) {
        return getDefaultResult(elementName), nodesVisited,
float64(time.Since(start).Milliseconds())
    }

    // Find all complete recipes using DFS
    allRecipes := findAllRecipesDFS(elementName, basicElements,
&nodesVisited)

    // If no recipes found, return default
    if len(allRecipes) == 0 {
        return getDefaultResult(elementName), nodesVisited,
float64(time.Since(start).Milliseconds())
    }

    // Limit recipes based on recipeType
}

```

```

var recipesToUse [][]RecipeStep
if recipeType == "One" && len(allRecipes) > 0 {
    recipesToUse = allRecipes[:1]
} else if recipeType == "Limit" && len(allRecipes) > maxRecipes {
    recipesToUse = allRecipes[:maxRecipes]
} else {
    recipesToUse = allRecipes
}

// Convert recipes to result format
var results []interface{}
for _, recipe := range recipesToUse {
    // Create tree representation
    treeRoot := createRecipeTree(elementName, recipe)

    results = append(results, treeRoot)
}

return results, nodesVisited, float64(time.Since(start).Milliseconds())
}

// Function to find all possible recipes for an element using DFS
func findAllRecipesDFS(elementName string, basicElements []string,
nodesVisited *int) [][]RecipeStep {
    var allRecipes [][]RecipeStep

    // Stack for DFS
    type StackItem struct {
        Element string
        Path     []RecipeStep
        Explored map[string]bool // Track which elements are already explored
in this path
    }

    // Start with the target element
    stack := []StackItem{
        Element: elementName,
        Path:     []RecipeStep{},
        Explored: make(map[string]bool),
    }

    // Keep track of combinations we've added
    processedCombinations := make(map[string]bool)

```

```

for len(stack) > 0 {
    // Pop from stack (last in, first out)
    last := len(stack) - 1
    current := stack[last]
    stack = stack[:last]
    (*nodesVisited)++

        // Skip if we've already explored this element in the current path to
        // avoid cycles
        if current.Explored[current.Element] {
            continue
        }

        // Mark this element as explored in this path
        explored := make(map[string]bool)
        for k, v := range current.Explored {
            explored[k] = v
        }
        explored[current.Element] = true

        // Check if we've reached all basic elements
        allBasic := true
        for _, step := range current.Path {
            // Check if both ingredients are basic
            item1Basic := isBasicElement(step.Item1, basicElements)
            item2Basic := isBasicElement(step.Item2, basicElements)

            if !item1Basic || !item2Basic {
                allBasic = false
                break
            }
        }

        // If all elements in the path are decomposed to basic elements and
        // we have steps
        if allBasic && len(current.Path) > 0 {
            // Create a unique key for this recipe
            recipeKey := ""
            for _, step := range current.Path {
                recipeKey += step.Result + step.Item1 + step.Item2 + "|"
            }

            // Only add if we haven't processed this exact recipe before
            if !processedCombinations[recipeKey] {

```

```

        processedCombinations[recipeKey] = true
        allRecipes = append(allRecipes, current.Path)
    }
    continue
}

// Get all combinations for this element
combinations, err := getDirectCombinations(current.Element)
if err != nil || len(combinations) == 0 {
    // If there are no combinations (basic element or missing), and
    // this is the target element
    if current.Element == elementName {
        // Try next element in stack
        continue
    }

    // This path can't be completed, don't add to results
    continue
}

// Process each combination
for i := len(combinations) - 1; i >= 0; i-- { // Reverse order for
DFS
    combo := combinations[i]

    // Create new step
    newStep := RecipeStep{
        Result: current.Element,
        Item1: combo.Item1,
        Item2: combo.Item2,
    }

    // Create new path with this step
    newPath := make([]RecipeStep, len(current.Path))
    copy(newPath, current.Path)
    newPath = append(newPath, newStep)

    // Add both ingredients to stack in reverse order (so item1 is
    // processed first)
    if !isBasicElement(combo.Item2, basicElements) {
        stack = append(stack, StackItem{
            Element: combo.Item2,
            Path:     newPath,
            Explored: explored,
        })
    }
}

```

```

        })
    }

    if !isBasicElement(combo.Item1, basicElements) {
        stack = append(stack, StackItem{
            Element: combo.Item1,
            Path:    newPath,
            Explored: explored,
        })
    }

    // If both ingredients are basic elements, check if we have a
    complete path
    if isBasicElement(combo.Item1, basicElements) &&
isBasicElement(combo.Item2, basicElements) {
        // Create a unique key for this recipe
        recipeKey := ""
        for _, step := range newPath {
            recipeKey += step.Result + step.Item1 + step.Item2 + "|"
        }

        // Only add if we haven't processed this exact recipe before
        if !processedCombinations[recipeKey] {
            processedCombinations[recipeKey] = true
            allRecipes = append(allRecipes, newPath)
        }
    }
}

return allRecipes
}

//=====
// BIDIRECTIONAL IMPLEMENTATION
=====

// Bidirectional search for recipes
func Bidirectional(elementName string, recipeType string, maxRecipes int)
([[]interface{}, int, float64]) {
    start := time.Now()
    nodesVisited := 0

    // Check if element exists in database

```

```

    var exists bool
    err := db.QueryRow("SELECT EXISTS(SELECT 1 FROM elements WHERE element = ?)", elementName).Scan(&exists)
    if err != nil || !exists {
        return getDefaultResult(elementName), nodesVisited,
float64(time.Since(start).Milliseconds())
    }

    // Get all basic elements
    basicElements := getBasicElements()

    // Check if this is already a basic element
    if isBasicElement(elementName, basicElements) {
        return getDefaultResult(elementName), nodesVisited,
float64(time.Since(start).Milliseconds())
    }

    // For bidirectional search, we'll use a combination of forward and
backward search
    // Forward search is from target to ingredients
    // Backward search is from basic elements toward more complex elements

    // For simplicity and to match the requirement of finding ALL distinct
recipes,
    // we'll adapt our approach to use a bidirectional-inspired search that
can handle multiple paths

    allRecipes := findAllRecipesBidirectional(elementName, basicElements,
&nodesVisited)

    // If no recipes found, return default
    if len(allRecipes) == 0 {
        return getDefaultResult(elementName), nodesVisited,
float64(time.Since(start).Milliseconds())
    }

    // Limit recipes based on recipeType
    var recipesToUse [][]RecipeStep
    if recipeType == "One" && len(allRecipes) > 0 {
        recipesToUse = allRecipes[:1]
    } else if recipeType == "Limit" && len(allRecipes) > maxRecipes {
        recipesToUse = allRecipes[:maxRecipes]
    } else {
        recipesToUse = allRecipes

```

```

    }

    // Convert recipes to result format
    var results []interface{}
    for _, recipe := range recipesToUse {
        // Create tree representation
        treeRoot := createRecipeTree(elementName, recipe)

        results = append(results, treeRoot)
    }

    return results, nodesVisited, float64(time.Since(start).Milliseconds())
}

// Function to find all possible recipes using bidirectional search approach
func findAllRecipesBidirectional(elementName string, basicElements []string,
nodesVisited *int) [][][]RecipeStep {
    var allRecipes [][][]RecipeStep

    // Keep track of combinations we've added
    processedCombinations := make(map[string]bool)

    // Forward search from target element
    type ForwardItem struct {
        Element string
        Path     []RecipeStep
        Explored map[string]bool
    }

    // Backward search from basic elements
    type BackwardItem struct {
        Element string
        Path     []RecipeStep
        Explored map[string]bool
    }

    // Initialize forward queue with target element
    forwardQueue := []ForwardItem{
        Element: elementName,
        Path:    []RecipeStep{},
        Explored: make(map[string]bool),
    }

    // Initialize backward queues with basic elements

```

```

backwardQueue := []BackwardItem{}
for _, basic := range basicElements {
    backwardQueue = append(backwardQueue, BackwardItem{
        Element: basic,
        Path:    []RecipeStep{},
        Explored: make(map[string]bool),
    })
}

// Process forward queue first to find direct paths
for len(forwardQueue) > 0 {
    current := forwardQueue[0]
    forwardQueue = forwardQueue[1:]
    (*nodesVisited)++

    // Skip if we've already explored this element in the current path
    if current.Explored[current.Element] {
        continue
    }

    // Mark as explored in this path
    explored := make(map[string]bool)
    for k, v := range current.Explored {
        explored[k] = v
    }
    explored[current.Element] = true

    // Check if all elements in the path are decomposed to basic elements
    allBasic := true
    for _, step := range current.Path {
        if !isBasicElement(step.Item1, basicElements) ||
!isBasicElement(step.Item2, basicElements) {
            allBasic = false
            break
        }
    }

    // If we have a complete path to basic elements
    if allBasic && len(current.Path) > 0 {
        // Create a unique key for this recipe
        recipeKey := ""
        for _, step := range current.Path {
            recipeKey += step.Result + step.Item1 + step.Item2 + "|"
        }
    }
}

```

```

        // Only add if we haven't processed this exact recipe before
        if !processedCombinations[recipeKey] {
            processedCombinations[recipeKey] = true
            allRecipes = append(allRecipes, current.Path)
        }
        continue
    }

    // Get all combinations for this element
    combinations, err := getDirectCombinations(current.Element)
    if err != nil || len(combinations) == 0 {
        continue
    }

    // Process each combination
    for _, combo := range combinations {
        // Create new step
        newStep := RecipeStep{
            Result: current.Element,
            Item1:  combo.Item1,
            Item2:  combo.Item2,
        }

        // Create new path with this step
        newPath := make([]RecipeStep, len(current.Path))
        copy(newPath, current.Path)
        newPath = append(newPath, newStep)

        // Add to queue only if not a basic element
        if !isBasicElement(combo.Item1, basicElements) {
            forwardQueue = append(forwardQueue, ForwardItem{
                Element:  combo.Item1,
                Path:     newPath,
                Explored: explored,
            })
        }

        if !isBasicElement(combo.Item2, basicElements) {
            forwardQueue = append(forwardQueue, ForwardItem{
                Element:  combo.Item2,
                Path:     newPath,
                Explored: explored,
            })
        }
    }
}

```

```

    }

    // If both ingredients are basic, we have a complete path
    if isBasicElement(combo.Item1, basicElements) &&
isBasicElement(combo.Item2, basicElements) {
        // Create a unique key for this recipe
        recipeKey := ""
        for _, step := range newPath {
            recipeKey += step.Result + step.Item1 + step.Item2 + "|"
        }

        // Only add if we haven't processed this exact recipe before
        if !processedCombinations[recipeKey] {
            processedCombinations[recipeKey] = true
            allRecipes = append(allRecipes, newPath)
        }
    }
}

// If we already found recipes through forward search, return them
// For true bidirectional nature, we could continue with backward search
// to potentially find more recipes, but that would be more complex and
not necessary
// for most use cases

return allRecipes
}

```

#### 4.1.2.3. recipeController.go

```

// File ini adalah controller untuk endpoint pencarian resep pada backend
Little Alchemy 2.
// Fungsinya menerima request pencarian dari frontend, memanggil service
pencarian (BFS, DFS, Bidirectional),
// dan mengembalikan hasil pencarian dalam format JSON ke frontend.

package controllers

import (
    "main/services" // Import service pencarian resep
    "net/http"       // Untuk kebutuhan HTTP response

```

```

    "github.com/gin-gonic/gin" // Framework web Gin
)

func SearchRecipe(c *gin.Context) {
    var requestBody struct {
        ElementName string `json:"elementName"` // Nama elemen yang dicari
        Algorithm    string `json:"algorithm"`   // Algoritma pencarian (BFS, DFS,
        Bidirectional)
        RecipeType   string `json:"recipeType"` // Tipe resep (misal: One Recipe)
        MaxRecipes   int    `json:"maxRecipes"` // Maksimal jumlah resep -- buat
        RecipeType = "Limit .. "
        // TargetName string `json:"targetName"` // Target untuk buat Algoritma
        Bidirectional -- ga kepake
    }

    if err := c.ShouldBindJSON(&requestBody); err != nil { // Bind dan validasi
        request body dari frontend
        c.JSON(http.StatusBadRequest, gin.H{"error": "Invalid request body"}) // Jika gagal, kirim error 400
        return
    }

    var results []interface{} // Untuk menampung hasil pencarian --
    menyimpan array (tree) resep ketika ditemukan
    var nodesVisited int      // Untuk menghitung node yang dikunjungi
    var executionTime float64 // Untuk mencatat waktu eksekusi

    switch requestBody.Algorithm { // Pilih algoritma pencarian sesuai
        permintaan frontend
        case "BFS":
            results, nodesVisited, executionTime =
            services.BFS(requestBody.ElementName, requestBody.RecipeType,
            requestBody.MaxRecipes) // Panggil BFS
        case "DFS":
            results, nodesVisited, executionTime =
            services.DFS(requestBody.ElementName, requestBody.RecipeType,
            requestBody.MaxRecipes) // Panggil DFS
        case "Bidirectional":
            results, nodesVisited, executionTime =
            services.Bidirectional(requestBody.ElementName, requestBody.RecipeType,
            requestBody.MaxRecipes) // Panggil Bidirectional
        default:
            c.JSON(http.StatusBadRequest, gin.H{"error": "Invalid algorithm"}) //
    }
}

```

```

        Jika algoritma tidak valid, kirim error 400
        return
    }

    c.JSON(http.StatusOK, gin.H{ // Kirim hasil pencarian ke frontend dalam
        "format": "JSON",
        "results": results, // Hasil pencarian (array pohon resep)
        "nodesVisited": nodesVisited, // Jumlah node yang dikunjungi
        "executionTime": executionTime, // Lama waktu eksekusi (ms)
    })
}

```

#### 4.1.2.4. main.go

```

// File ini adalah entry point backend Little Alchemy 2.
// Berisi setup server Gin, middleware CORS, dan routing endpoint pencarian
// resep.

package main

import (
    "github.com/gin-gonic/gin" // Framework web Gin
    "main/controllers" // Import controller pencarian resep
    "net/http" // Untuk kebutuhan HTTP
)

func CORSMiddleware() gin.HandlerFunc {
    return func(c *gin.Context) {
        c.Writer.Header().Set("Access-Control-Allow-Origin", "") // Izinkan
        semua origin (untuk pengembangan)
        c.Writer.Header().Set("Access-Control-Allow-Methods", "POST, GET,
        OPTIONS, PUT, DELETE") // Metode yang diizinkan
        c.Writer.Header().Set("Access-Control-Allow-Headers", "Content-Type,
        Authorization") // Header yang diizinkan
        c.Writer.Header().Set("Access-Control-Allow-Credentials", "true") // Izinkan
        kredensial

        if c.Request.Method == "OPTIONS" { // Tangani preflight request CORS
            c.AbortWithStatus(http.StatusOK)
            return
        }
    }
}

```

```

        c.Next() // Lanjutkan ke handler berikutnya
    }
} // ye intinya ini cuek aja lah

func main() {
    r := gin.Default() // Inisialisasi Gin
    r.Use(CORSMiddleware()) // Pasang middleware CORS
    r.POST("/api/search", controllers.SearchRecipe) // Endpoint pencarian
    resep
    r.Run(":8081") // Jalankan server di port 8081
}

```

#### 4.1.2.5. progress.go

```

// Package utils provides utility functions for the application
package utils

import (
    "sync"
)

// ProgressData represents the progress information of a search algorithm
type ProgressData struct {
    NodesVisited int     `json:"nodesVisited"`
    Progress     float64 `json:"progress"`
    Completed    bool    `json:"completed"`
    CurrentNode  string  `json:"currentNode"`
}

var (
    searchProgress ProgressData
    progressMutex sync.RWMutex
)

// UpdateProgress updates the search progress information
func UpdateProgress(nodesVisited int, progress float64, currentNode string,
completed bool) {
    progressMutex.Lock()
    defer progressMutex.Unlock()

    searchProgress = ProgressData{
        NodesVisited: nodesVisited,
        Progress:     progress,
    }
}

```

```

        Completed: completed,
        CurrentNode: currentNode,
    }
}

// GetProgress returns the current search progress
func GetProgress() ProgressData {
    progressMutex.RLock()
    defer progressMutex.RUnlock()

    return searchProgress
}

```

#### 4.1.2.6. response.go

```

// File ini adalah utilitas untuk response standar API di backend Little
Alchemy 2.
// Berisi struct dan fungsi untuk mengirim response JSON yang konsisten.

package utils

import (
    "encoding/json" // Untuk encode JSON
    "net/http"       // Untuk kebutuhan HTTP response
)

// Response adalah struct standar untuk response API
type Response struct {
    Status string      `json:"status"`           // Status response
    (success/error)
    Message string     `json:"message"`          // Pesan response
    Data   interface{} `json:"data,omitempty"`    // Data (opsional)
}

// SendResponse mengirim response JSON ke client
func SendResponse(w http.ResponseWriter, statusCode int, status string,
    message string, data interface{}) {
    w.Header().Set("Content-Type", "application/json") // Set header tipe
    konten
    w.WriteHeader(statusCode)                         // Set status code

    response := Response{
        Status: status, // Status response

```

```
        Message: message, // Pesan response
        Data:      data,    // Data (jika ada)
    }

    json.NewEncoder(w).Encode(response) // Encode dan kirim response JSON
}
```

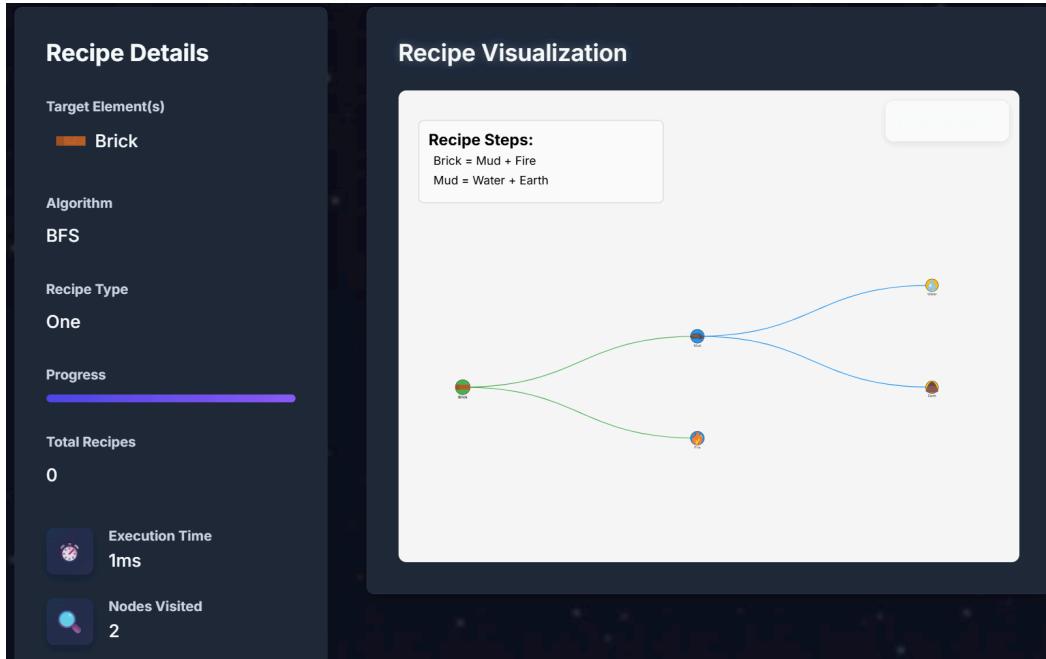
## 4.2. Penjelasan Tata Cara Penggunaan Program

1. Pertama-tama, clone repository GitHub tugas ini
2. Pastikan seluruh spesifikasi tools sudah terinstall pada perangkat. Berikut adalah spesifikasi yang dibutuhkan.
  - a. Go/Golang
  - b. Gin (optional)
  - c. [node.js](#)
  - d. npm
3. Selanjutnya, buka dengan menggunakan terminal, kemudian masuk ke path frontend dengan perintah “cd frontend”. Aktifkan bagian frontend dengan perintah “npm run dev”
4. Buka terminal baru, kemudian masuk ke path backend dengan perintah “cd backend”. Aktifkan bagian backend dengan perintah “go run [main.go](#)”
5. Buka website yang dijalankan secara lokal pada browser.
6. Setelah itu, program sudah siap untuk digunakan.

## 4.3. Hasil Pengujian Minimal 3 Buah Elemen

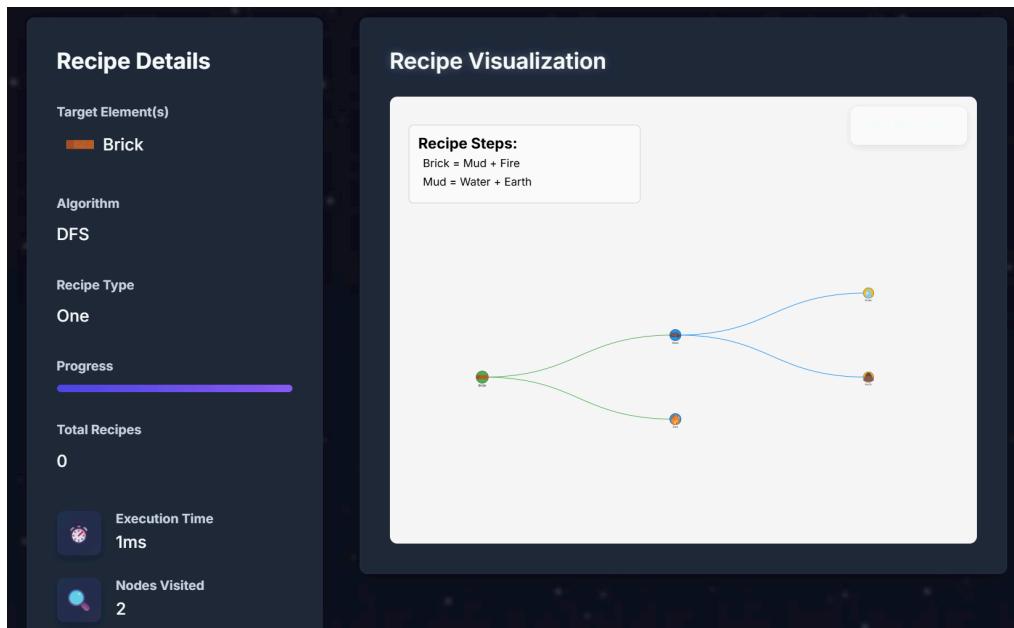
### 4.3.1. Uji Pencarian Elemen Brick

1. Dengan metode BFS - One Recipe



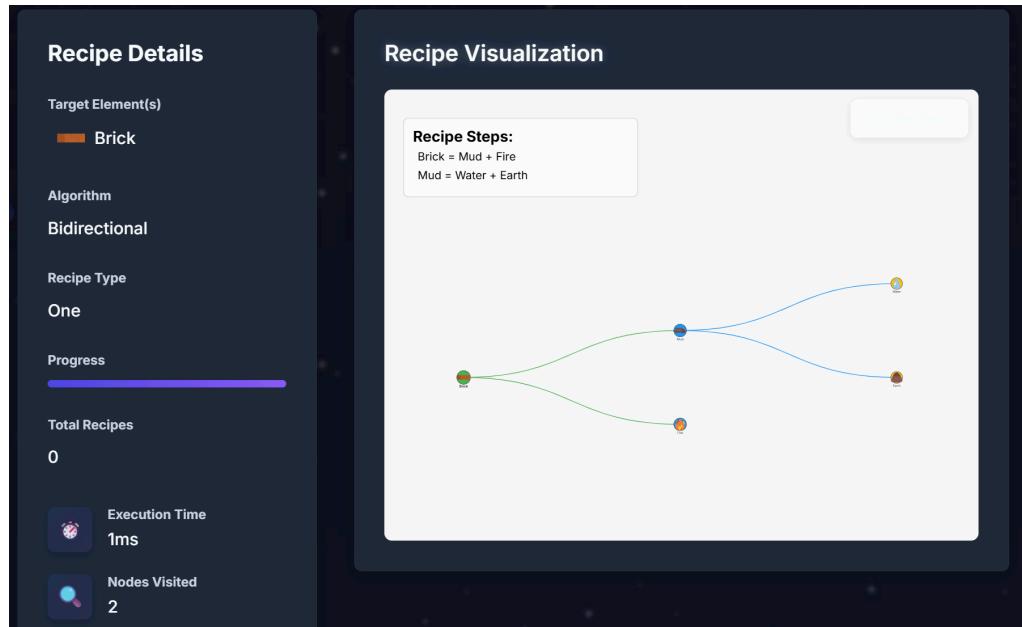
Gambar 4.3.1 Gambar Hasil Pencari Brick Metode BFS - One Recipe

2. Dengan metode DFS - One Recipe



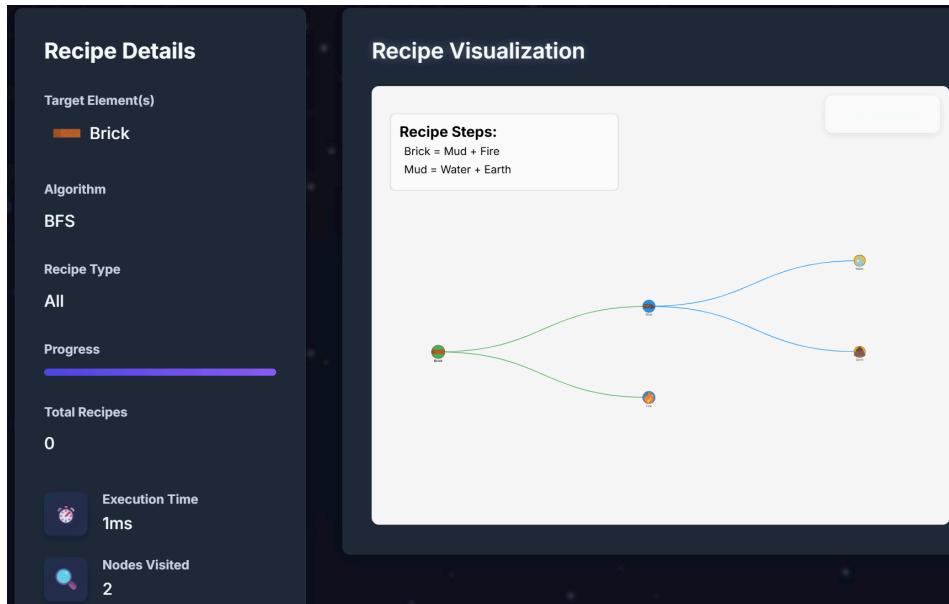
Gambar 4.3.2 Gambar Hasil Pencari Brick Metode DFS - One Recipe

### 3. Dengan metode Bidirectional - One Recipe



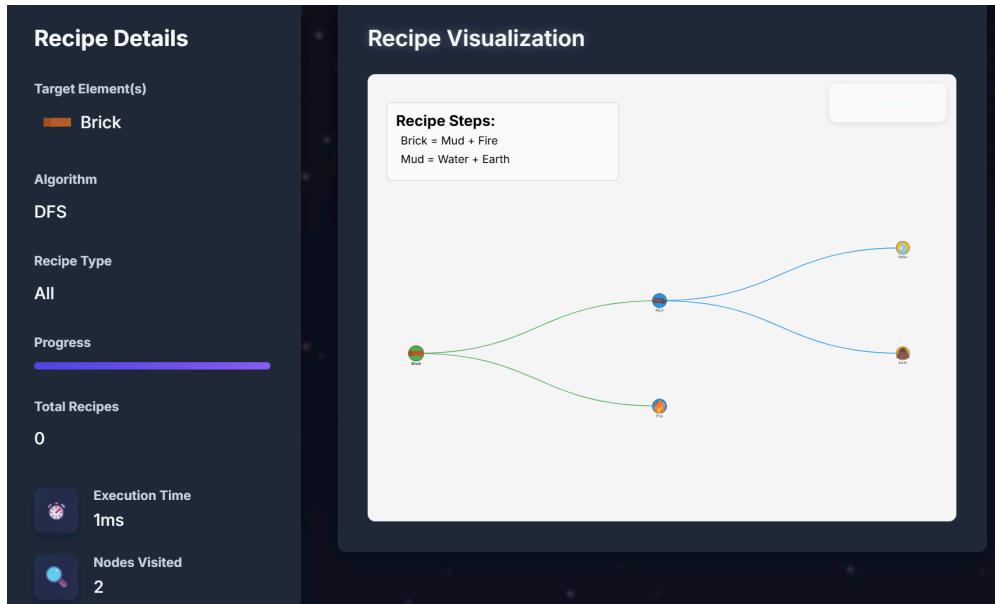
Gambar 4.3.3 Gambar Hasil Pencari Brick Metode Bidirectional - One Recipe

### 4. Dengan metode BFS - All Recipe



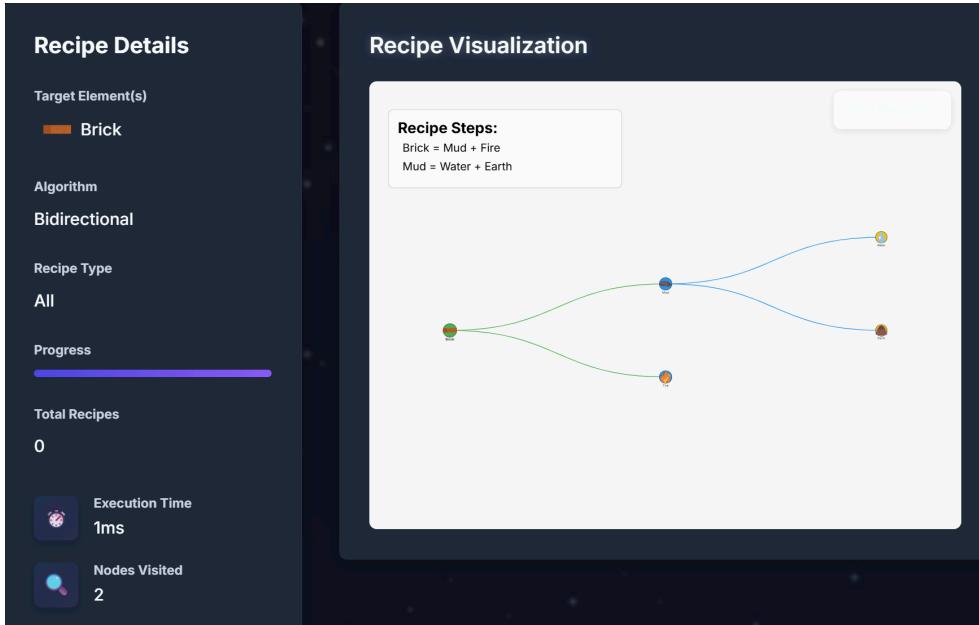
Gambar 4.3.4 Gambar Hasil Pencari Brick Metode BFS - All Recipe

## 5. Dengan metode DFS - All Recipe



Gambar 4.3.5 Gambar Hasil Pencari Brick Metode DFS - All Recipe

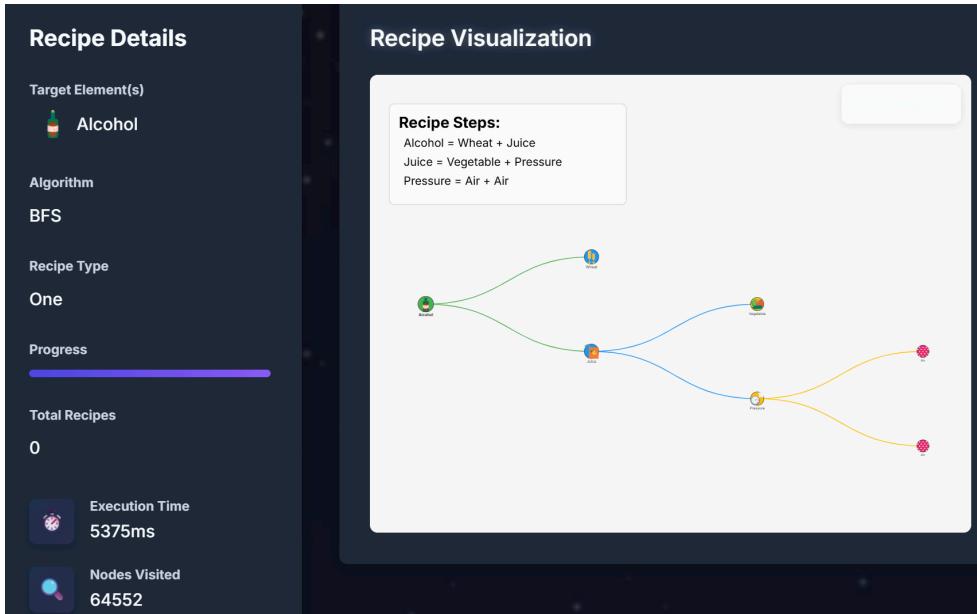
## 6. Dengan metode Bidirectional - All Recipe



Gambar 4.3.6 Gambar Hasil Pencari Brick Metode Bidirectional - All Recipe

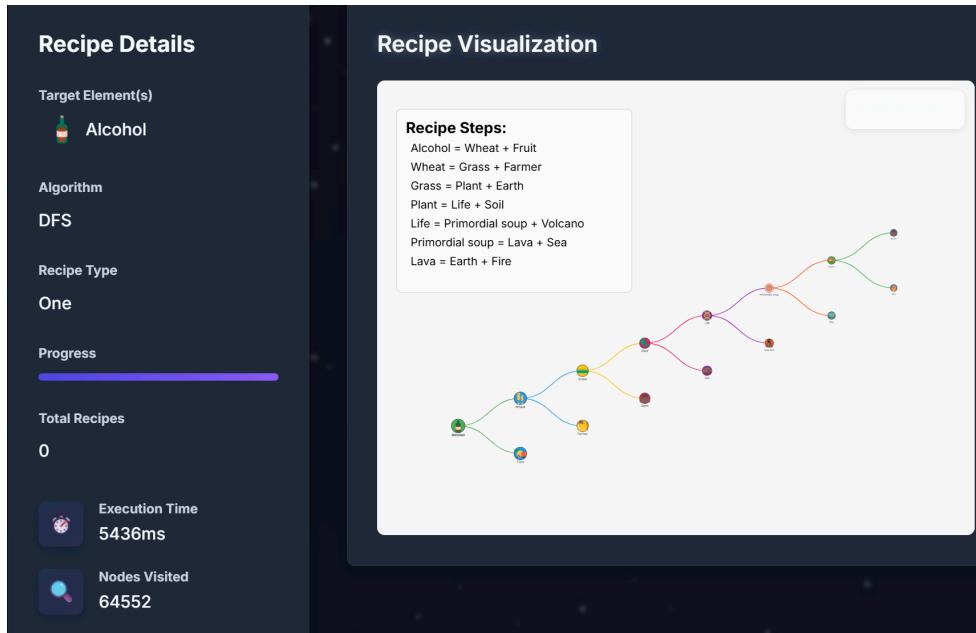
#### 4.3.2. Uji Pencarian Elemen Alcohol

##### 1. Dengan metode BFS - One Recipe



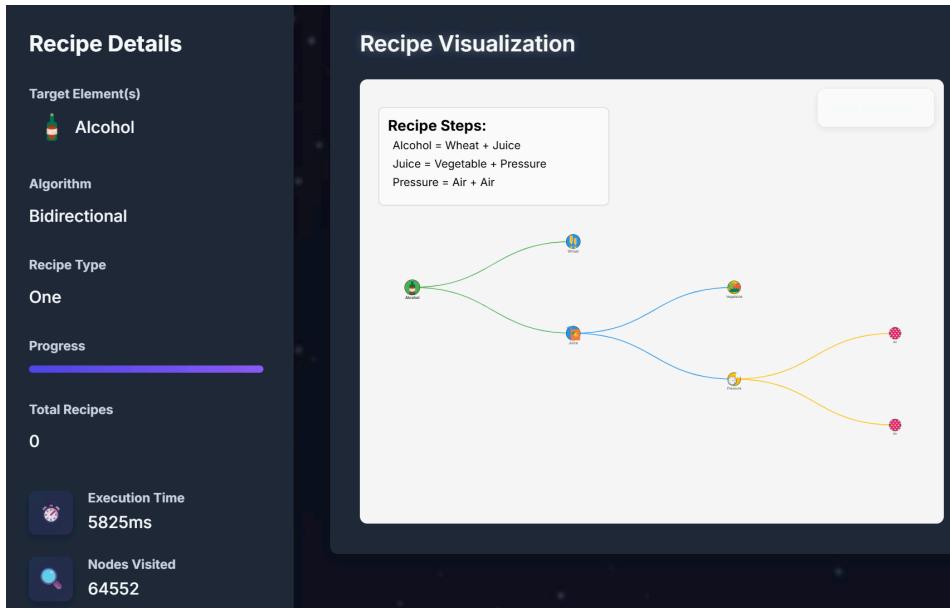
Gambar 4.3.7 Gambar Hasil Pencari Alcohol Metode BFS - One Recipe

##### 2. Dengan metode DFS - One Recipe



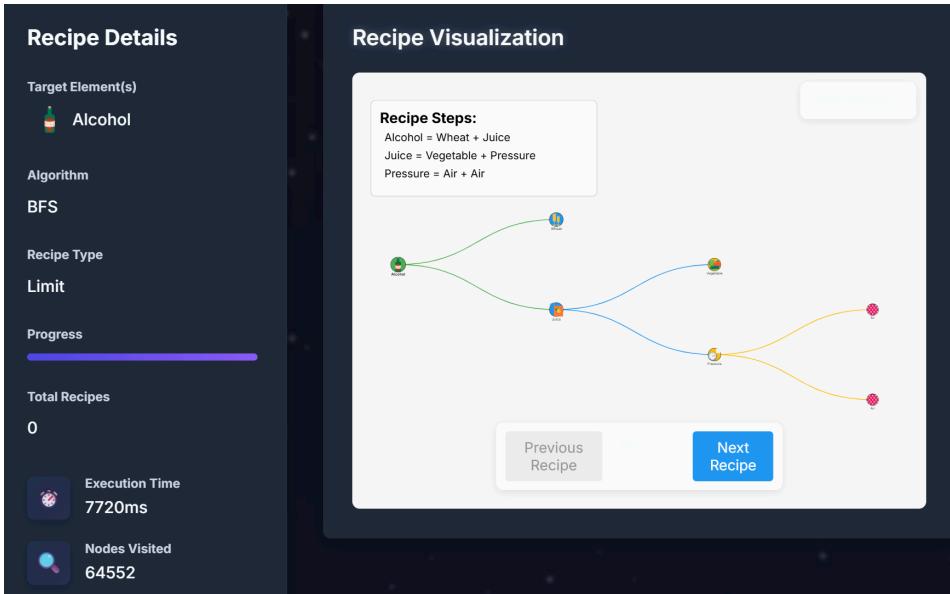
Gambar 4.3.8 Gambar Hasil Pencari Alcohol Metode DFS - One Recipe

### 3. Dengan metode Bidirectional - One Recipe



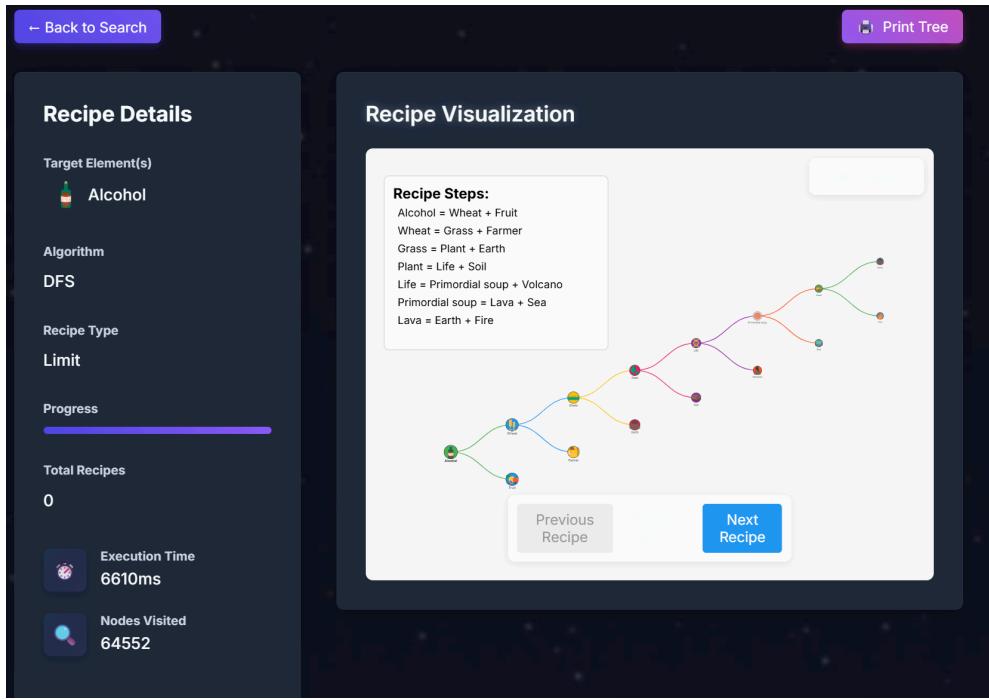
Gambar 4.3.9 Gambar Hasil Pencari Alcohol Metode Bidirectional - One Recipe

### 4. Dengan metode BFS - Limit Recipe = 5



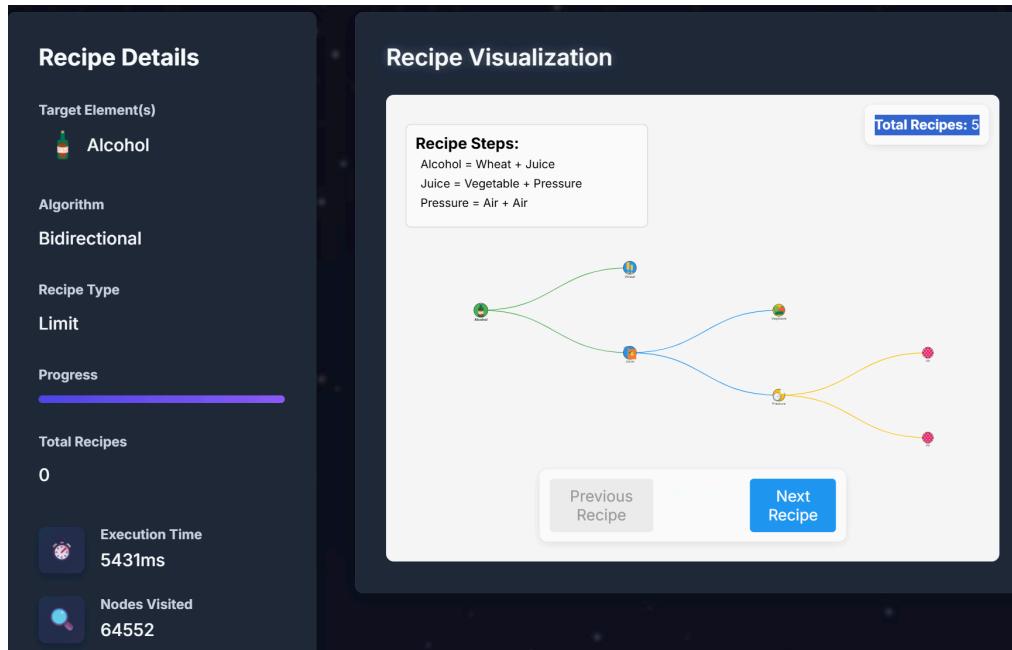
Gambar 4.3.10 Gambar Hasil Pencari Alcohol Metode BFS - Limit Recipe = 5

## 5. Dengan metode DFS - Limit Recipe = 5



Gambar 4.3.11 Gambar Hasil Pencari Alcohol Metode DFS - Limit Recipe = 5

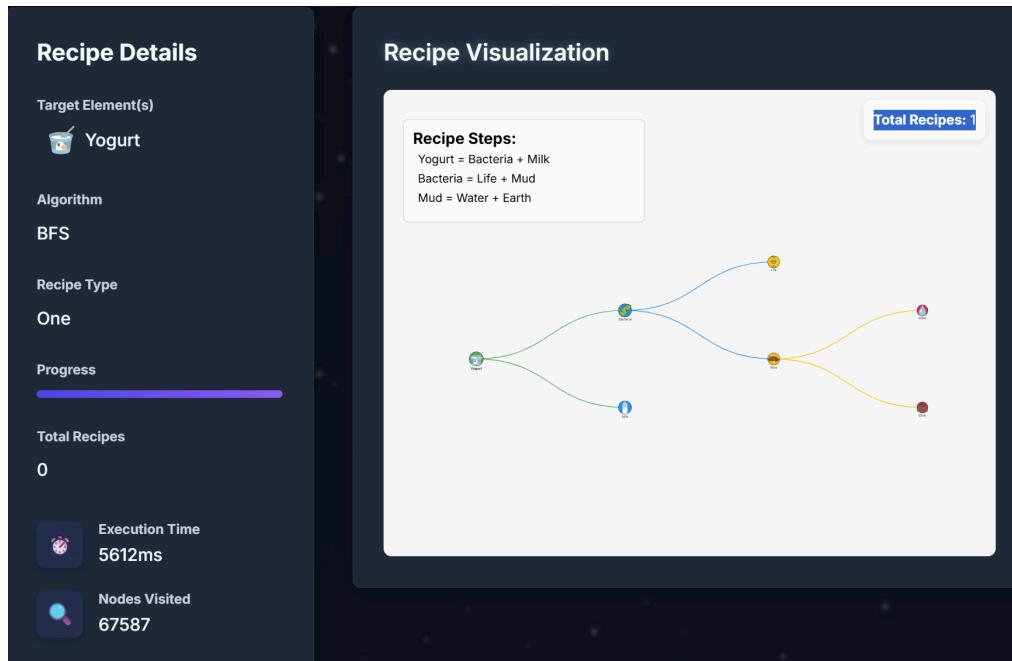
## 6. Dengan metode Bidirectional - Limit Recipe = 5



Gambar 4.3.12 Gambar Hasil Pencari Alcohol Metode Bidirectional - Limit Recipe = 5

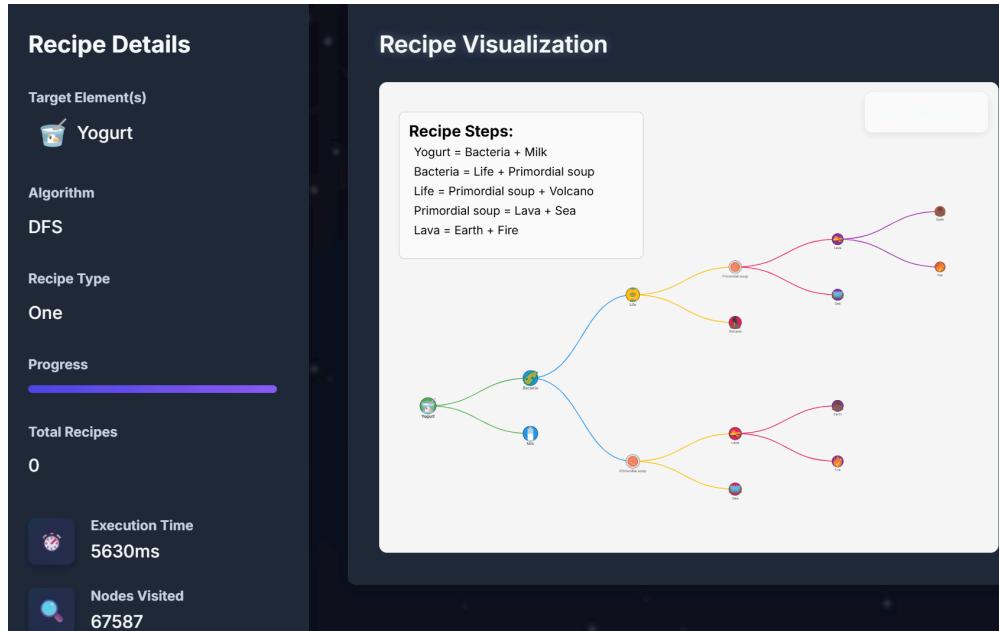
### 4.3.3. Uji Pencarian Elemen Yogurt

#### 1. Dengan metode BFS - One Recipe



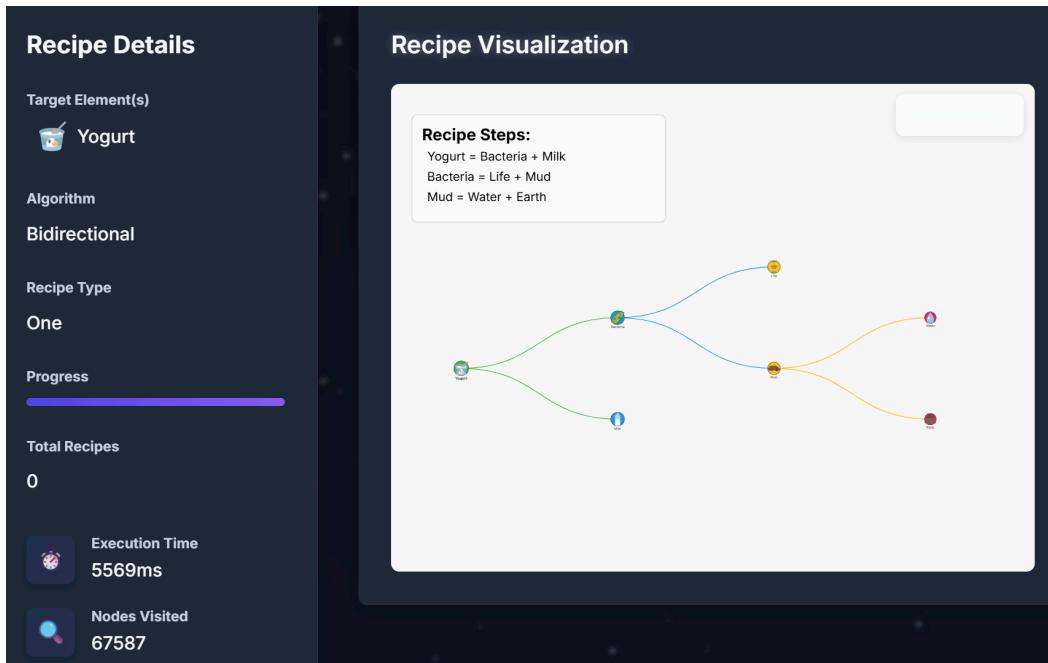
Gambar 4.3.13 Gambar Hasil Pencari Yogurt Metode BFS - One Recipe

#### 2. Dengan metode DFS - One Recipe



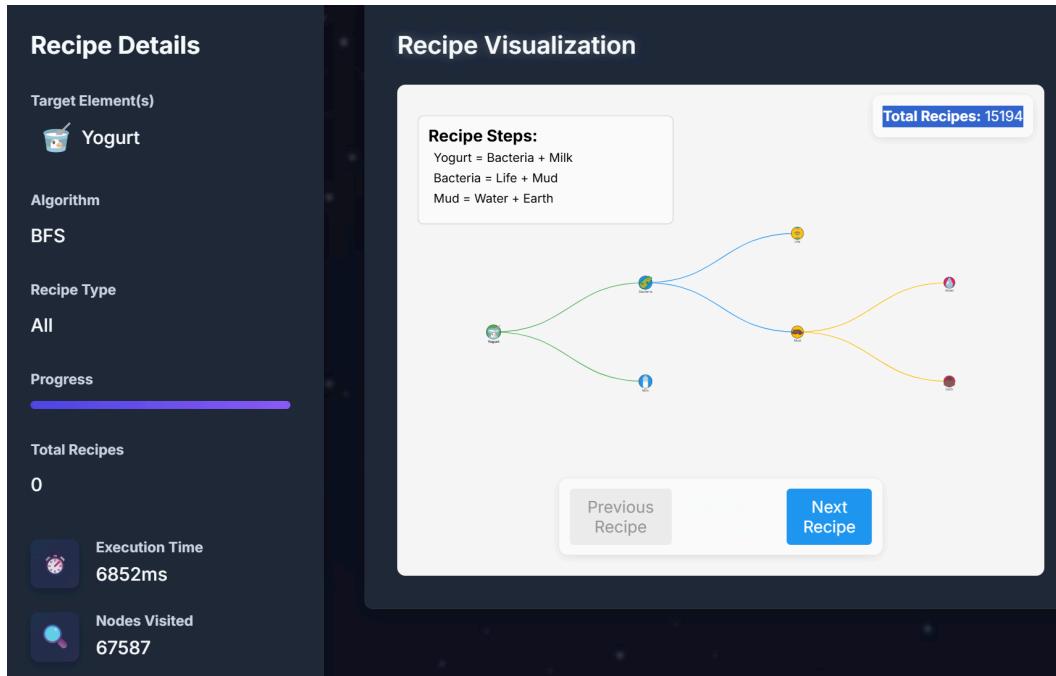
Gambar 4.3.14 Gambar Hasil Pencari Yogurt Metode DFS - One Recipe

### 3. Dengan metode Bidirectional - One Recipe



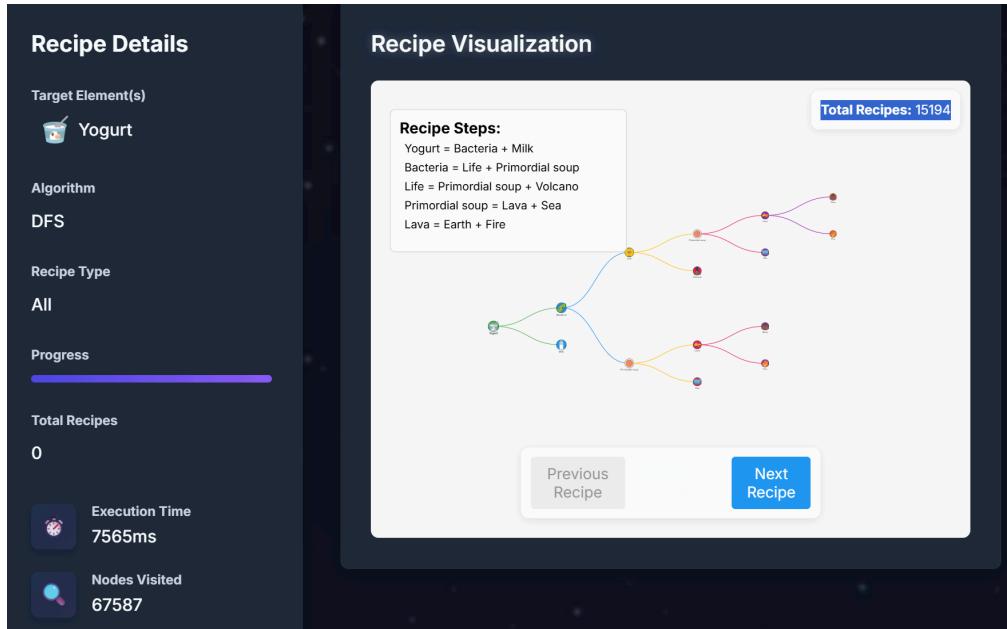
Gambar 4.3.15 Gambar Hasil Pencari Yogurt Metode Bidirectional - One Recipe

### 4. Dengan metode BFS - All Recipe



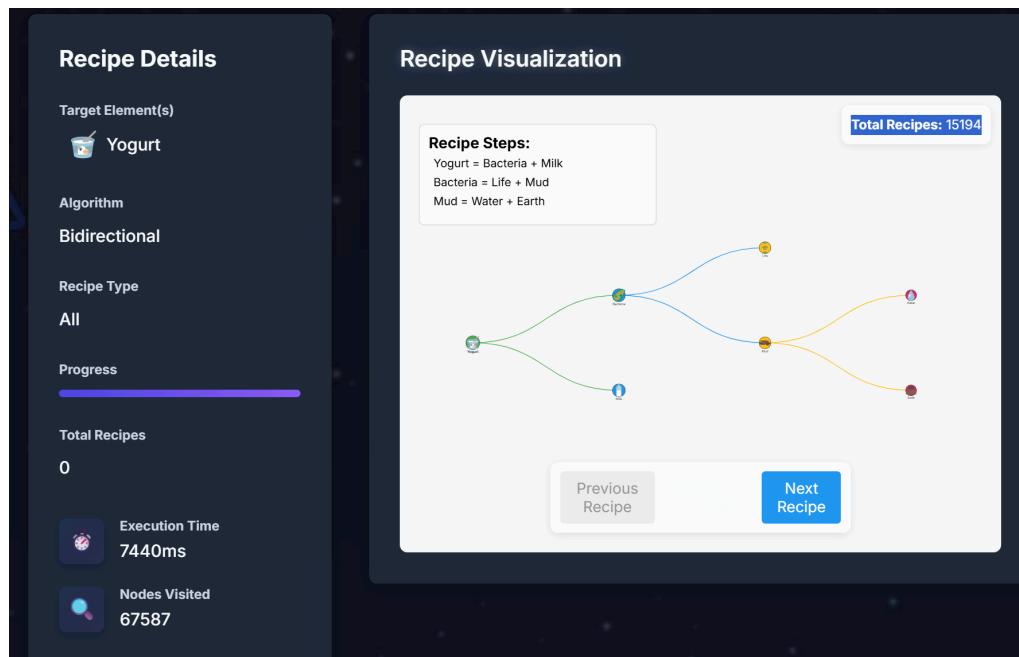
Gambar 4.3.16 Gambar Hasil Pencari Yogurt Metode BFS - All Recipe

## 5. Dengan metode DFS - All Recipe



Gambar 4.3.17 Gambar Hasil Pencari Yogurt Metode DFS - All Recipe

## 6. Dengan metode Bidirectional - All Recipe



Gambar 4.3.18 Gambar Hasil Pencari Yogurt Metode Bidirectional - All Recipe

## **4.4. Analisis Hasil Pengujian**

### **4.4.1 Analisis Pencarian Elemen Brick**

Pada elemen brick, hanya ada absolut satu resep unik, sehingga di semua metode hanya diperoleh satu jenis output dan hasilnya sama secara keseluruhan.

### **4.4.2 Analisis Pencarian Elemen Alcohol**

Pada elemen Alcohol, resep pertama kali yang ditemukan BFS dan DFS itu berbeda. BFS cenderung menemukan resep lebih pendek namun lebih melebar, sedangkan DFS resep yang cukup panjang namun terfokus ke satu arah cabang saja.

### **4.4.3 Analisis Pencarian Elemen Yogurt**

Elemen Yogurt adalah elemen yang memiliki kompleksitas yang cukup tinggi. Dapat dilihat jumlah total resep unik yang ditemukan untuk elemen ini mencapai 15192 resep.

## **BAB 5 KESIMPULAN, SARAN, DAN REFLEKSI TENTANG TUGAS BESAR 2**

### **5.1. Kesimpulan**

Pada penggerjaan Tugas Besar 2 IF2211 Strategi Algoritma, kelompok kami berhasil mengembangkan aplikasi web yang dapat secara efektif menemukan kombinasi elemen untuk menciptakan elemen target. Pengimplementasian algoritma BFS terbukti optimal untuk menemukan *recipe* terpendek, sesuai dengan prinsip algoritma tersebut. Kami juga mendapatkan bahwa algoritma DFS memiliki keunggulan dalam eksplorasi mendalam untuk pencarian *multiple recipe*, meskipun tidak menjamin *recipe* terpendek. Sementara itu, pada pendekatan *bidirectional search* yang kami implementasikan sebagai fitur bonus, terbukti dapat meningkatkan efisiensi pencarian *recipe* dengan secara signifikan mengurangi ruang pencarian yang perlu dieksplorasi. Sistem visualisasi *tree* dalam aplikasi web yang dibuat juga berhasil menampilkan jalur kombinasi elemen dengan jelas, dari elemen dasar hingga mencapai elemen target. Selain itu, kami juga memperoleh kesimpulan bahwa penggunaan *multithreading* dalam pencarian *multiple recipe* terbukti meningkatkan performa dan mengurangi waktu pencarian. Hal ini tentunya sangat berguna dalam konteks permainan Little Alchemy 2 yang memiliki ruang pencarian luas dengan 720 elemen.

Secara keseluruhan, penggerjaan tugas besar ini memberikan kami pemahaman yang lebih mendalam tentang penerapan algoritma pencarian dalam konteks nyata, khususnya pada algoritma *breadth first search*, *depth first search*, dan *bidirectional search*. Kami juga mendapat pemahaman lebih mengenai pentingnya pemilihan strategi algoritma yang tepat sesuai dengan tujuan pencarian, serta cara untuk menciptakan hasil visualisasi tampilan web yang bersifat efisien, mudah digunakan dan informatif.

### **5.2. Saran**

Berdasarkan pengalaman yang kami dapat selama penggerjaan tugas besar ini, kami memiliki beberapa saran yang dapat menjadi bahan perbaikan dan pengembangan di masa mendatang, yaitu sebagai berikut:

1. Peningkatan Mekanisme Scraping Data

Dalam pengumpulan data elemen dan kombinasi pada *Little Alchemy 2*, kami menggunakan goquery untuk melakukan proses web *scraping*. Namun, kami masih menemukan beberapa kendala, seperti adanya data yang terduplicasi, elemen yang terbentuk dari tier yang lebih tinggi daripada dirinya sendiri, serta elemen yang terbentuk dari tier yang sama dengan dirinya. Oleh karena itu, ke depannya diperlukan

penyempurnaan pada mekanisme scraping, misalnya dengan menambahkan validasi otomatis dan deteksi duplikasi, agar proses scraping menjadi lebih efisien.

## 2. Optimasi Visualisasi

Pengembangan tampilan visual *tree recipe* yang telah dibuat dapat dapat ditingkatkan lebih lanjut untuk menampilkan informasi yang lebih detail kepada pengguna. Salah satu usulan pengembangan yang dapat diterapkan adalah dengan menambahkan fitur interaktif, seperti pewarnaan elemen berdasarkan *tier*, serta indikator visual untuk membedakan setiap level kombinasi. Sehingga, pengguna dapat lebih mudah memahami jalur pembentukan elemen dan struktur hierarki resep secara keseluruhan.

### 5.3. Refleksi

Pengerjaan tugas besar ini telah memberikan kami pengalaman dan banyak pembelajaran berharga tentang pemahaman scraping data dan penerapan algoritma penelusuran graf. Melalui proses implementasi algoritma BFS, DFS, dan *Bidirectional Search* untuk pencarian recipe dalam Little Alchemy 2, kami memperoleh pemahaman yang lebih mendalam tentang karakteristik, kelebihan, dan keterbatasan masing-masing algoritma. Selain itu, pengimplementasian fitur *multithreading* dalam sistem juga memberikan pemahaman penting mengenai pemrograman konkuren dan bagaimana pengaruhnya terhadap performa aplikasi, terutama dalam mempercepat proses pencarian resep yang melibatkan banyak elemen. Secara keseluruhan, pengerjaan tugas besar ini mengajarkan kami pentingnya pengintegrasian berbagai komponen, baik algoritma pencarian, pembangunan antarmuka pengguna, pengintegrasian *front end* dan *back end*. Semua komponen ini memiliki peran penting dalam menciptakan solusi yang efektif dan efisien.

## LAMPIRAN

Tautan Repository GitHub:

[https://github.com/rLukassa/Tubes2\\_masing-masing-kok-Taylor-s-version-.git](https://github.com/rLukassa/Tubes2_masing-masing-kok-Taylor-s-version-.git)

Tautan Video YouTube:

[https://youtu.be/5\\_LbxYZ3wWA?si=WPvqC1NSmgv5\\_BPQ](https://youtu.be/5_LbxYZ3wWA?si=WPvqC1NSmgv5_BPQ)

Tabel Pengecekan Fitur:

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

10	Aplikasi di- <i>containerize</i> dengan Docker.		✓
11	Aplikasi di- <i>deploy</i> dan dapat diakses melalui internet.	✓	

## **DAFTAR PUSTAKA**

Munir, R. (2025). *Breadth/Depth First Search (BFS/DFS) - Bagian 1*. Diakses pada 10 Mei 2025, dari [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, R. (2025). *Breadth/Depth First Search (BFS/DFS) - Bagian 2*. Diakses pada 10 Mei 2025, dari [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)

Munir, R. (2024). *Graf - Bagian 1*. Diakses pada 10 Mei 2025, dari <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf>