

**LAPORAN TUGAS BESAR 2
IF2211 STRATEGI ALGORITMA**



oleh:

Kelompok 59 - K03

Rhio Bimo Prakoso Sugiyanto 13523123

M. Kinan Arkansyaddad 13523152

Fachriza Ahmad Setiyono 13523162

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025**

DAFTAR ISI

DAFTAR ISI.....	2
BAB I	
DESKRIPSI TUGAS.....	3
BAB II	
LANDASAN TEORI.....	5
A. Traversal Graf.....	5
B. Algoritma BFS (Breadth-First Search).....	5
C. Algoritma DFS (Depth-First Search).....	7
D. Aplikasi Web.....	8
BAB III	
ANALISIS PEMECAHAN MASALAH.....	10
A. Mapping Persoalan My Little Alchemist.....	10
B. Langkah-langkah Pemecahan Persoalan.....	10
a. DFS.....	11
b. BFS.....	12
C. Fitur Fungsional dan Arsitektur Aplikasi Web yang Dibangun.....	13
D. Ilustrasi Kasus.....	16
BAB IV	
IMPLEMENTASI DAN PENGUJIAN.....	17
A. Spesifikasi Teknis Program.....	17
a. Scraper.....	18
b. Model.....	20
c. DB.....	31
d. API.....	33
a. Search Page.....	39
B. Penjelasan Tata Cara Penggunaan Program.....	42
C. Hasil Pengujian.....	43
D. Analisis Hasil.....	47
BAB V	
KESIMPULAN, SARAN, DAN REFLEKSI.....	49
A. Kesimpulan.....	49
B. Saran.....	49
C. Refleksi.....	50
LAMPIRAN.....	51
DAFTAR PUSTAKA.....	53

BAB I

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 II

LANDASAN TEORI

A. Traversal Graf

Traversal graf adalah proses sistematis untuk mengunjungi setiap simpul dalam sebuah graf, biasanya dengan tujuan mengeksplorasi struktur, mencari informasi tertentu, atau memecahkan permasalahan yang berkaitan dengan jalur atau keterhubungan antar simpul. Traversal memungkinkan algoritma untuk menelusuri graf secara menyeluruh, baik melalui pendekatan kedalaman terlebih dahulu (Depth-First Search/DFS) maupun lebar terlebih dahulu (Breadth-First Search/BFS).

Pada dasarnya, traversal dilakukan dengan memilih simpul awal, lalu mengikuti hubungan antar simpul (sisi) untuk berpindah dari satu simpul ke simpul lainnya. Proses ini dilakukan sambil menjaga jejak simpul yang sudah dikunjungi guna menghindari pengulangan dan siklus tak berujung.

Traversal graf dapat diterapkan pada graf tak berarah maupun berarah, dan baik terhubung maupun tak terhubung, tergantung kebutuhan. Struktur data seperti stack, queue, rekursi, dan matriks ketetanggaan atau daftar ketetanggaan sering digunakan untuk mendukung proses ini.

Traversal memiliki peran penting dalam berbagai aplikasi, mulai dari analisis jaringan sosial, pencarian rute optimal, deteksi siklus, hingga topological sorting. Metode traversal yang dipilih—apakah DFS atau BFS—akan sangat bergantung pada karakteristik graf dan tujuan eksplorasi.

Dengan demikian, traversal graf merupakan fondasi utama dari algoritma BFS dan DFS, yang masing-masing mengadopsi strategi traversal berbeda: BFS menjelajahi simpul secara melebar (level demi level), sedangkan DFS menyelam lebih dalam ke jalur sebelum kembali (backtrack).

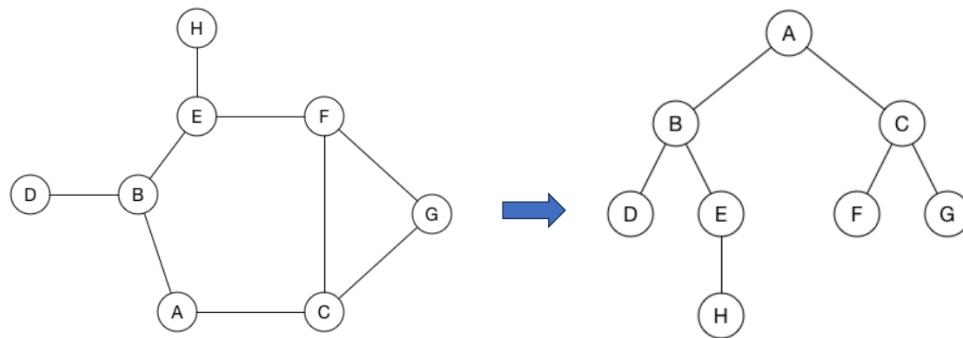
B. Algoritma BFS (Breadth-First Search)

Algoritma BFS memecahkan persoalan pencarian graf langkah demi langkah dengan strategi “per-level” atau “melebar” secara sistematis:

1. mengambil semua pilihan di satu tingkat (jarak) sebelum melangkah ke tingkat berikutnya
2. berharap bahwa menyisir simpul berjarak k terlebih dahulu akan segera menemukan lintasan terpendek dalam jumlah sisi

BFS mengasumsikan bahwa perlu memeriksa semua simpul pada jarak terdekat sebelum melangkah lebih jauh, sehingga solusi terpendek (bila bobot sisi seragam) akan terdeteksi di level paling atas.

Breadth-First Search (BFS) merupakan algoritma penelusuran graf yang mengandalkan struktur data queue (antrean) dengan prinsip First-In, First-Out (FIFO). Algoritma ini dimulai dengan menandai simpul awal sebagai telah dikunjungi, lalu memasukkannya ke dalam antrean. Selanjutnya, simpul pertama dalam antrean diambil, dan seluruh tetangganya yang belum dikunjungi akan dimasukkan ke antrean dan ditandai sebagai dikunjungi agar tidak diproses ulang. Proses ini diulang terus hingga antrean kosong atau simpul tujuan telah ditemukan.



Gambar 3. Contoh Algoritma DFS

Secara umum, implementasi BFS melibatkan matriks ketetanggaan $A = [a_{ij}]$ berukuran $n \times n$, dimana $a_{ij} = 1$ menunjukkan bahwa simpul i dan j saling terhubung, dan $a_{ij} = 0$ bila tidak. Selain itu, digunakan queue q untuk menyimpan antrian simpul yang menunggu diproses, serta array boolean *visited* untuk mencatat simpul yang sudah diperlakukan.

Kelebihan utama BFS terletak pada kemampuannya menemukan jalur terpendek dalam graf tak berbobot, karena algoritma ini mengeksplorasi simpul yang paling dekat terlebih dahulu. Oleh karena itu, BFS sangat bermanfaat untuk pencarian jalur pada peta, penyelesaian puzzle, serta berbagai permasalahan graf lainnya. BFS juga bisa digunakan untuk mengetahui keterhubungan antar simpul dan menemukan komponen terhubung dalam graf.

Meski demikian, BFS memiliki beberapa kekurangan. Salah satunya adalah konsumsi memori yang cukup besar, terutama pada graf dengan banyak simpul dan sisi, karena semua simpul yang belum dikunjungi harus disimpan dalam antrean. Selain itu,

BFS kurang efisien pada graf yang sangat dalam, atau pada permasalahan yang membutuhkan penelusuran mendalam terlebih dahulu sebelum mencapai solusi.

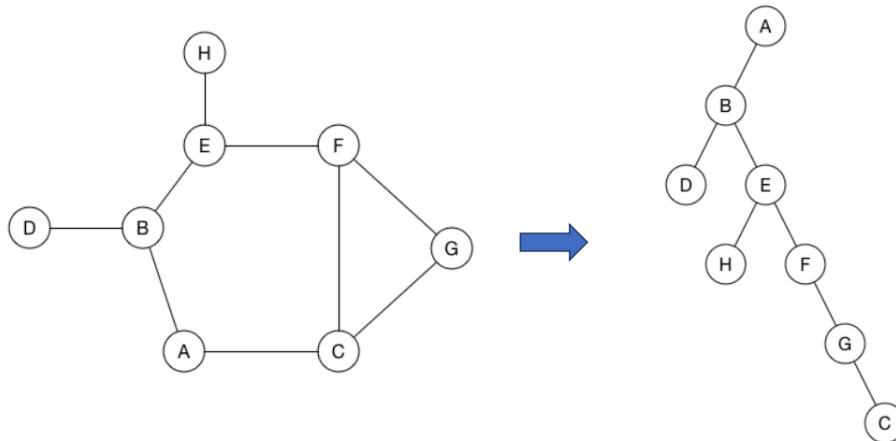
C. Algoritma DFS (Depth-First Search)

Algoritma DFS memecahkan persoalan pencarian atau penjelajahan graf secara langkah demi langkah dengan strategi “sejauh-jauhnya sebelum kembali” (backtracking):

1. mengambil pilihan berikutnya untuk dieksplorasi sedalam mungkin saat itu
2. berharap bahwa menelusuri satu cabang sampai tuntas sebelum pindah ke cabang lain akan menemukan solusi dengan cepat

Pada setiap langkah, algoritma ini mengunjungi simpul yang belum dikunjungi dan bersebelahan dengan simpul yang sedang diperiksa. Proses ini berlanjut sampai menemui jalan buntu, yaitu saat tidak ada lagi simpul tetangga yang belum dikunjungi. Kemudian, algoritma akan melakukan backtracking ke simpul sebelumnya untuk mencoba mengunjungi simpul yang belum dikunjungi dari sana.

DFS akan berhenti setelah mundur kembali ke simpul awal dan menemui jalan buntu. Pada titik ini, semua simpul dan komponen graf yang sama dengan simpul awal sudah dikunjungi. Jika masih ada simpul yang belum dikunjungi, pencarian harus dimulai lagi dari salah satu simpul di komponen graf lainnya. Dengan demikian, DFS juga dapat digunakan untuk mengidentifikasi komponen terhubung dalam graf tak berarah.



Gambar 4. Contoh Algoritma DFS

(sumber: <http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf>)

Algoritma Depth-First Search (DFS) menggunakan struktur data stack dengan prinsip Last-In, First-Out (LIFO). Stack ini berfungsi untuk mencatat simpul-simpul yang akan dikunjungi kembali jika pencarian menemui jalan buntu. DFS juga bisa

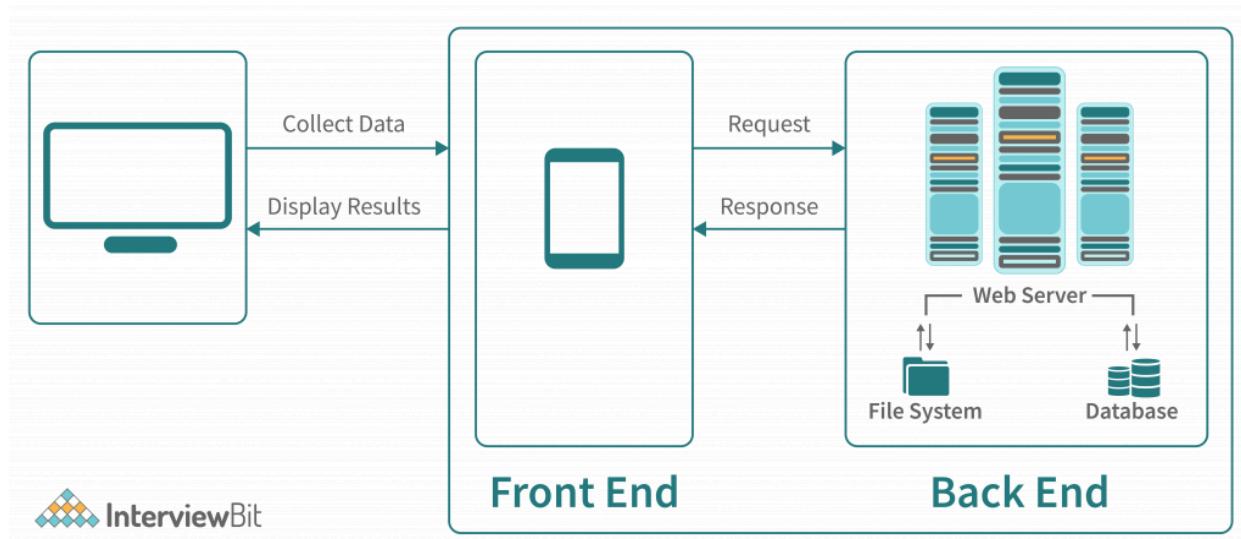
diimplementasikan secara rekursif, yang secara tidak langsung menggunakan call stack untuk menyimpan jejak simpul yang telah dikunjungi.

Setiap kali DFS menelusuri simpul baru, simpul tersebut dimasukkan ke dalam stack. Jika tidak ada jalur lanjutan dari simpul tersebut, algoritma akan kembali (backtrack) ke simpul sebelumnya yang masih memiliki tetangga yang belum dikunjungi.

Salah satu kelebihan DFS adalah efisiensi penggunaan memorinya pada graf dengan cabang yang tidak terlalu banyak. Selain itu, DFS efektif untuk menjelajahi simpul hingga kedalaman maksimum dengan cepat. DFS juga sangat berguna untuk sejumlah permasalahan seperti deteksi siklus dalam graf, topological sort, serta pencarian komponen terhubung. Pendekatan ini juga fleksibel dan dapat dimodifikasi dengan mudah untuk berbagai variasi masalah yang membutuhkan eksplorasi mendalam terlebih dahulu.

Namun, DFS juga memiliki beberapa keterbatasan. DFS tidak cocok untuk mencari jalur terpendek pada graf yang tidak berbobot, karena tidak memprioritaskan simpul yang paling dekat. Di graf yang besar dan kompleks, DFS bisa menelusuri terlalu dalam, sehingga kurang efisien. Jika menggunakan pendekatan rekursif, ada risiko stack overflow jika kedalaman penelusuran terlalu besar.

D. Aplikasi Web



Gambar 5. Struktur Aplikasi Web

(sumber: <https://www.interviewbit.com/blog/web-application-architecture/>)

Arsitektur Aplikasi Web adalah struktur dasar yang menentukan bagaimana berbagai komponen dalam sebuah aplikasi web—termasuk frontend, backend, server, dan

database—berinteraksi melalui jaringan untuk memberikan layanan kepada pengguna akhir. Arsitektur ini tidak hanya mencakup desain visual aplikasi, tetapi juga bagaimana data diproses, dikirim, dan disajikan secara efisien dan aman. Tujuan utama dari arsitektur aplikasi web adalah untuk memastikan bahwa aplikasi dapat berjalan dengan lancar, berskala, mudah dikelola, dan aman dari ancaman siber.

Arsitektur web pada dasarnya mengikuti model client-server. Dalam model ini, klien (biasanya browser pengguna) mengirimkan permintaan (request) ke server melalui protokol HTTP/HTTPS. Server kemudian memproses permintaan tersebut—misalnya mengambil data dari database atau menjalankan logika—and mengembalikan hasilnya kepada klien dalam bentuk tanggapan (response), biasanya berupa halaman HTML atau data dalam format JSON. Klien kemudian menampilkan informasi tersebut kepada pengguna. Arsitektur client-server bersifat terpusat, artinya server bertindak sebagai otoritas utama yang mengatur dan melayani semua permintaan dari klien.

Arsitektur web terdapat berbagai jenis, seperti *Monolithic Architecture* yang seluruh logika aplikasi berjalan dalam satu unit, cocok untuk proyek kecil namun sulit diskalakan dan *Microservices Architecture* dimana web aplikasi dibagi menjadi layanan-layanan kecil yang independen, lebih fleksibel dan mudah dikembangkan secara terpisah. Ada juga *Serverless Architecture*, pengembang hanya menulis fungsi-fungsi tertentu tanpa mengelola server secara langsung. Infrastruktur dikelola oleh pihak ketiga seperti AWS Lambda atau Vercel Functions. *Single Page Applications (SPA)* merupakan arsitektur web yang hanya terdiri satu halaman HTML yang dimuat, dan konten lainnya dimuat secara dinamis menggunakan JavaScript serta komunikasi API, memungkinkan pengalaman pengguna yang lebih cepat dan interaktif.

BAB III

ANALISIS PEMECAHAN MASALAH

Pada bab ini akan dibahas langkah-langkah dan komponen utama dalam merancang sistem pencarian resep di website “My Little Alchemist”. Setiap poin menguraikan aspek penting mulai dari pemetaan masalah hingga penentuan strategi pencarian

A. Mapping Persoalan My Little Alchemist

Permasalahan inti dalam sistem My Little Alchemist adalah bagaimana menemukan jalur kombinasi untuk membentuk suatu elemen dari dua bahan dasar yang tersedia dalam permainan Little Alchemy 2. Tidak seperti graf tradisional di mana relasi hanya terjadi antar simpul tunggal, pada permainan ini, setiap elemen hanya dapat terbentuk dari pasangan spesifik dua elemen. Oleh karena itu, struktur graf konvensional yang bersifat satu-ke-satu tidak dapat secara langsung merepresentasikan jenis ketergantungan yang dibutuhkan.

Dalam konteks ini, setiap elemen hasil merupakan simpul (node) yang memiliki relasi dengan satu atau lebih pasangan elemen yang bisa menghasilkan elemen tersebut. Artinya, satu node dapat memiliki banyak jalur masuk (in-degree), namun masing-masing jalur terdiri dari dua simpul sekaligus sebagai bahan pembentuknya. Permasalahan menjadi lebih kompleks ketika satu bahan merupakan hasil dari kombinasi lain, sehingga jalur pencarian membentuk pohon resep yang bercabang-cabang dan mendalam.

Untuk menangani kompleksitas ini, diperlukan pendekatan pemodelan yang mampu mengakomodasi relasi dua-ke-satu, di mana setiap simpul tidak hanya terhubung dari satu simpul lain, tetapi dari pasangan. Oleh karena itu, struktur graf yang digunakan harus menyimpan informasi semua kemungkinan kombinasi dua elemen yang dapat membentuk sebuah hasil, bukan hanya hubungan linear antar simpul.

Dengan memahami bahwa bentuk relasi di permainan bersifat komposit dan rekursif, proses pemetaan ini menjadi fondasi dalam membangun sistem pencarian resep. Pemetaan masalah ini mengarah langsung pada kebutuhan untuk membuat graf yang mampu menyimpan relasi pasangan, serta pohon pencarian yang dapat memvisualisasikan jalur pembentukan elemen dari bahan dasar ke hasil akhir secara menyeluruh.

B. Langkah-langkah Pemecahan Persoalan

Masalah utama yang ingin diselesaikan dalam proyek ini adalah menemukan cara membuat suatu elemen dalam permainan Little Alchemy 2, yang diperoleh dari penggabungan dua elemen berbeda. Tidak seperti sistem graf biasa yang hanya

menghubungkan satu simpul ke simpul lain, dalam permainan ini, setiap elemen hanya bisa dibentuk dari kombinasi spesifik dua elemen, sehingga dibutuhkan struktur graf khusus yang mampu merepresentasikan relasi dua elemen sebagai prasyarat terbentuknya satu elemen baru.

Langkah awal dalam penyelesaian masalah ini adalah membangun representasi data yang memuat seluruh elemen dan resep dalam permainan. Data tersebut diperoleh melalui proses web scraping dari situs wiki resmi Little Alchemy 2, lalu disusun dalam struktur graf di mana setiap simpul mewakili satu elemen hasil, dan simpul tersebut memiliki satu atau lebih pasangan elemen yang bisa digunakan untuk membentuknya.

Graf ini kemudian direpresentasikan sebagai tabel data relasional, yang memetakan setiap elemen hasil ke dalam daftar pasangan bahan pembentuknya. Dengan pendekatan ini, setiap elemen dapat dilacak secara sistematis berdasarkan kombinasi bahan yang valid.

Agar pencarian resep dapat divisualisasikan dengan baik, dibentuk struktur data `ElementNode`. Setiap `ElementNode` berisi informasi tentang elemen tersebut, induk simpul, dan daftar resepnya dengan struktur data `RecipeNode`. Selain itu, node ini juga menyimpan jumlah resep yang valid dari seluruh resepnya.

Dengan struktur seperti ini, sistem dapat menampilkan relasi antar elemen secara hierarkis, memperlihatkan dengan jelas asal kombinasi tiap elemen, serta mendukung visualisasi dan penelusuran yang bersifat rekursif.

Setelah graf dan pohon pencarian terbentuk, pengguna dapat memberikan input berupa elemen target yang ingin diketahui resepnya, serta memilih metode pencarian: Breadth-First Search (BFS) atau Depth-First Search (DFS). Terdapat juga opsi "multiple recipe" agar pengguna dapat melihat beberapa alternatif kombinasi pembentuk.

Sistem kemudian akan menjalankan algoritma pencarian sesuai pilihan pengguna, menelusuri jalur-jalur dalam graf, dan mencatat simpul-simpul yang dikunjungi, kombinasi resep yang ditemukan, serta waktu eksekusi algoritma. Hasil pencarian akan divisualisasikan dalam bentuk pohon resep, di mana setiap node menunjukkan elemen hasil, dan anak-anaknya merupakan pasangan bahan pembentuk. Informasi seperti jumlah node yang diproses dan durasi pencarian akan ditampilkan di antarmuka web agar pengguna dapat memahami prosesnya dengan lebih mudah.

a. DFS

Penjelajahan graf secara DFS umumnya bisa dilakukan secara iteratif menggunakan *stack* atau secara rekursif dengan pemanggilan fungsi berulang pada setiap node anak. Untuk kasus ini, metode rekursif digunakan.

Hal yang perlu diperhatikan pada kasus ini adalah DFS dilakukan dengan *multithreading*, sehingga pada setiap rekursi resep, bahan kedua tidak bisa menunggu untuk mengetahui kemungkinan resep valid yang dihasilkan dari bahan pertama. Ini menyulitkan pengaturan pembatasan jumlah anak resep pada metode *multi recipe*, sehingga jumlah resep yang dihasilkan dapat melebihi target yang telah ditentukan.

Proses rekursi DFS program ini dapat dijabarkan sebagai berikut:

- i. Cek elemen *input*, jika elemen adalah elemen dasar, maka kembalikan *ElementNode* berisi elemen tersebut.
- ii. Untuk setiap resep yang memungkinkan dari elemen tersebut:
 1. Jika jumlah resep yang ada pada akar pohon sama atau lebih dari target, maka keluar dari *loop*.
 2. Panggil fungsi lagi terhadap bahan pertama resep dalam *thread* lain, dan terhadap bahan kedua resep dalam *thread* saat itu.
- iii. Kembalikan *ElementNode*.

b. BFS

Penjelajahan graf secara BFS dilakukan dengan menjelajahi seluruh kemungkinan resep dari suatu elemen, dan menggunakan *queue* sebagai antrian elemen yang perlu dicek. BFS umumnya digunakan untuk mencari jarak terpendek, dimana penjelajahan bisa langsung diberhentikan setelah jalur yang valid ditemukan.

Untuk menunjang *multithreading* dalam BFS, maka pemanfaat *thread* baru dapat dilakukan di tiap pemrosesan elemen untuk memproses masing-masing resep. Hal ini memastikan BFS tetap berjalan secara benar dengan paralel.

Proses rekursi BFS program ini dapat dijabarkan sebagai berikut:

- i. Mulai dengan elemen target sebagai salah satu bahan resep dan masukkan ke *queue*
- ii. Untuk elemen terdepan *queue*:
 1. Ambil seluruh resep dari elemen tersebut
 2. Hitung jumlah resep dari node tersebut hingga ke *root*. Jika sudah melebihi target jumlah resep, maka berhenti.
 3. Proses kedua bahan dengan *multithreading* untuk mendapatkan resep valid dari bahan dan memasukkannya ke *queue*.
- iii. Potong cabang pohon yang tidak memiliki resep valid dan kembalikan *ElementNode* akar.

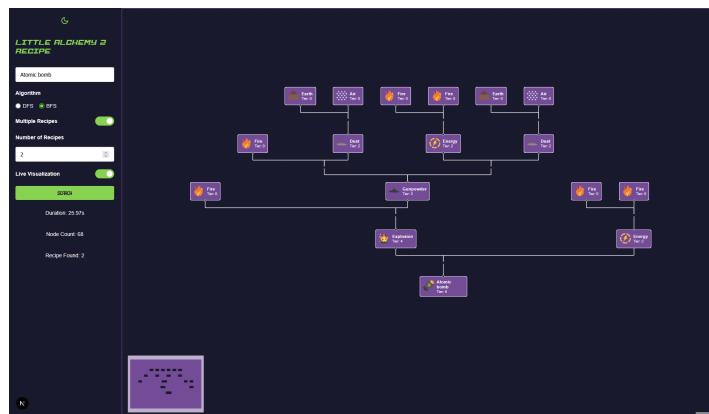
C. Fitur Fungsional dan Arsitektur Aplikasi Web yang Dibangun.

Dalam web kami, arsitektur yang digunakan berbasis model Client-Server. Klien dibangun menggunakan React melalui framework Next.js, yang bekerja sebagai Single Page Application (SPA). SPA memungkinkan halaman aplikasi dimuat satu kali, lalu semua interaksi dan navigasi dilakukan secara dinamis tanpa perlu memuat ulang seluruh halaman. Ketika pengguna melakukan pencarian resep, permintaan dikirimkan ke server melalui HTTP (menggunakan axios) atau melalui Server-Sent Events (SSE) untuk pencarian secara langsung (*live search*). Backend kemudian memproses permintaan tersebut dengan algoritma pencarian (DFS atau BFS) dan merespons dengan data struktur pohon yang divisualisasikan kembali di sisi klien. Arsitektur ini memungkinkan responsivitas yang tinggi serta pemrosesan yang efisien antara klien dan server.

Fitur-fitur fungsional yang terdapat pada website kami di antaranya:

1. Little Alchemy 2 Recipes Finder dan Recipes Tree Visualization

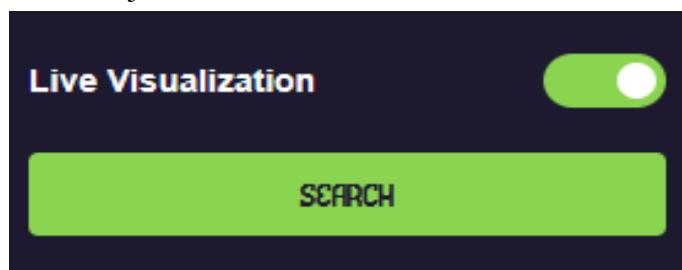
Fitur utama dari website ini adalah sebagai alat bantu untuk menemukan resep dari berbagai elemen dalam permainan Little Alchemy 2. Pengguna dapat memilih satu elemen dari daftar yang disediakan, lalu menentukan metode pencarian resep, yaitu menggunakan algoritma Depth First Search (DFS) atau Breadth First Search (BFS). Setelah memilih, pengguna cukup menekan tombol “Search”, dan sistem akan menghubungi backend API untuk memproses dan mencari cara-cara pembentukan elemen tersebut dari kombinasi elemen-elemen dasar. Hasilnya kemudian ditampilkan dalam bentuk visual berupa pohon resep (recipe tree), yang menggambarkan langkah-langkah kombinasi dari elemen dasar hingga membentuk elemen yang dipilih. Ini memberikan pengalaman yang intuitif dalam memahami struktur dan urutan kombinasi dalam permainan.



Gambar 6. Visualisasi Recipes Tree (Dark Mode)
(sumber: Arsip kelompok)

2. Live Update Visualization

Salah satu fitur dari website ini adalah dukungan untuk visualisasi pencarian resep secara langsung atau real-time menggunakan teknologi Server-Sent Events (SSE). Saat pengguna mengaktifkan opsi "Live Visualization", sistem akan membuka koneksi yang memungkinkan backend untuk mengirimkan data resep secara bertahap. Dengan ini, pengguna dapat melihat perkembangan visual pohon resep secara dinamis seiring proses pencarian berjalan, tanpa harus menunggu hasil akhir terlebih dahulu. Ini sangat membantu terutama untuk resep yang memiliki banyak kemungkinan kombinasi, serta memberikan pengalaman yang lebih interaktif dan transparan dalam melihat bagaimana sistem bekerja menemukan solusi.



Gambar 7. Tombol Live Visualization (Dark Mode)
(sumber: Arsip kelompok)

3. Minimap Recipes Tree

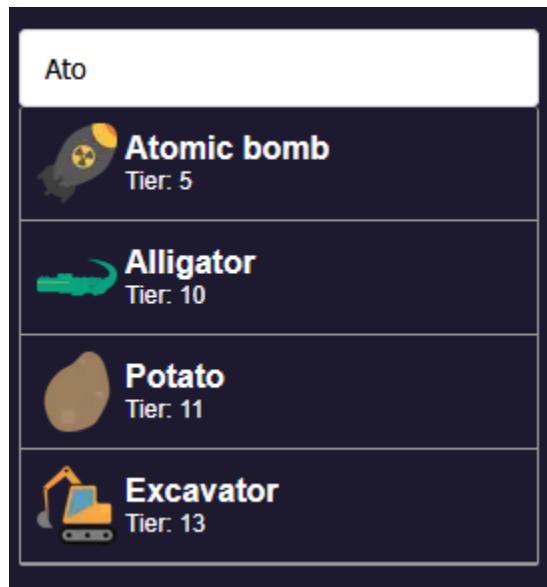
Visualisasi pohon resep yang ditampilkan menggunakan komponen *RecipeFlow* mendukung fitur minimap atau peta kecil yang mempermudah navigasi struktur pohon yang kompleks. Fitur ini sangat berguna ketika pohon resep memiliki banyak simpul dan cabang, karena pengguna dapat dengan mudah melihat posisi mereka dalam keseluruhan pohon dan menjelajahi setiap bagian dengan lebih cepat. Dengan kemampuan zoom, drag, dan minimap, pengguna tidak hanya mendapatkan informasi yang lengkap, tetapi juga kendali penuh atas bagaimana mereka ingin menjelajahi dan memahami hubungan antar elemen. Ini menjadikan eksplorasi data lebih efisien dan menyenangkan.



Gambar 8. Tampilan Minimap (Dark Mode)
(sumber: Arsip kelompok)

4. Autocomplete Text Input

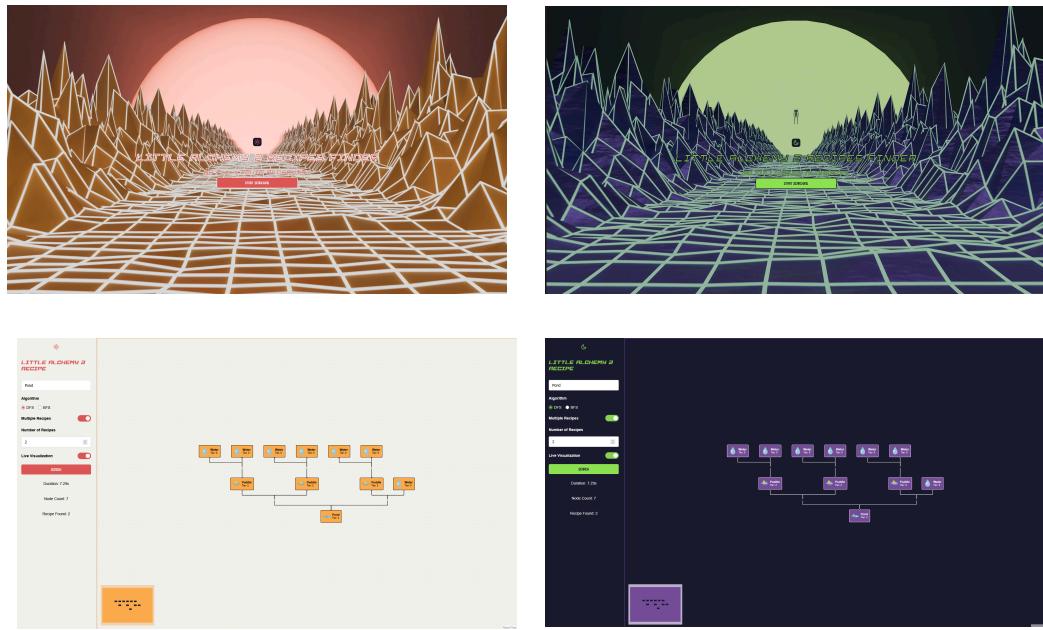
Untuk meningkatkan kenyamanan dan efisiensi pengguna, website ini menyediakan fitur input teks dengan autocomplete saat memilih elemen yang ingin dicari. Komponen input akan secara otomatis menyarankan nama-nama elemen yang tersedia berdasarkan daftar yang diambil dari backend. Saat pengguna mulai mengetik beberapa huruf, sistem akan menampilkan pilihan yang relevan, sehingga pengguna tidak perlu mengetik nama elemen secara lengkap. Fitur ini tidak hanya mempercepat proses pencarian, tetapi juga memastikan kesalahan input tidak terjadi (pengguna diwajibkan untuk memilih opsi yang ditampilkan untuk melakukan searching) dan meningkatkan aksesibilitas, terutama bagi pengguna yang tidak hafal seluruh nama elemen dalam permainan.



Gambar 9. Tampilan Autocomplete (Dark Mode)
(sumber: Arsip kelompok)

5. Light dan Dark Mode

Website ini juga dilengkapi dengan fitur light mode dan dark mode untuk memberikan fleksibilitas tampilan sesuai preferensi pengguna. Mode ini dapat diubah kapan saja menggunakan tombol toggle yang tersedia di bagian atas antarmuka. Semua elemen visual termasuk teks, latar belakang, tombol, dan bahkan video latar belakang akan menyesuaikan secara otomatis berdasarkan mode yang dipilih. Dalam mode gelap, misalnya, video latar akan berubah menjadi klip dengan nuansa gelap, sementara dalam mode terang akan menampilkan suasana yang lebih cerah. Fitur ini tidak hanya memperindah tampilan, tetapi juga meningkatkan kenyamanan mata, terutama saat digunakan dalam kondisi cahaya yang berbeda.



Gambar 10. Perbandingan Tampilan Dark Mode dan Light Mode
(sumber: Arsip kelompok)

D. Ilustrasi Kasus

Ilustrasi kasus Little Alchemy 2, Atomic Bomb:

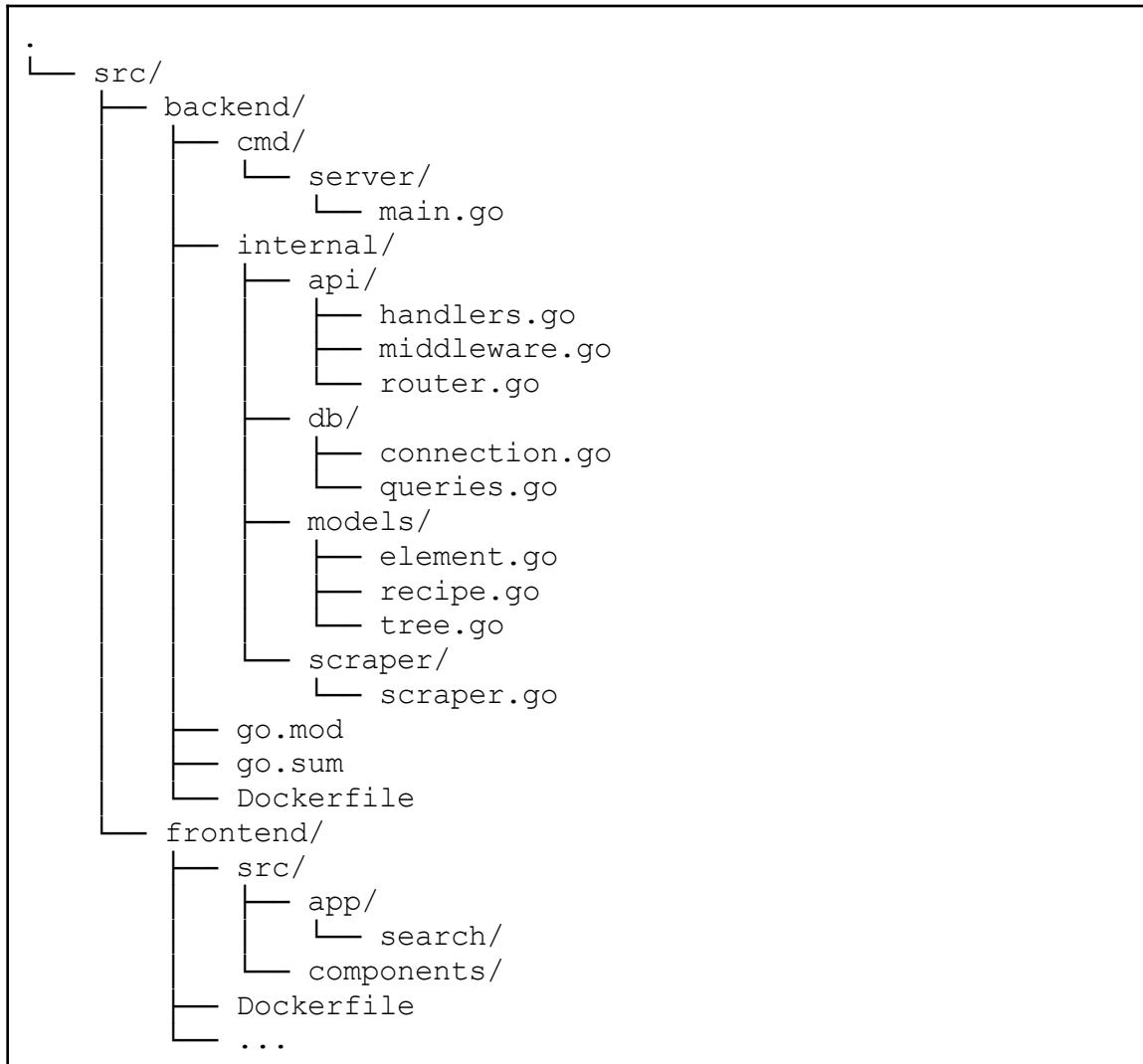
1. Pengguna ingin mencari 1 resep elemen Atomic Bomb
2. Pengguna memilih salah satu dari dua metode pencarian resep, yaitu DFS
3. Program menjalankan proses penjelajahan graf resep dengan DFS:
 - i. Cek elemen *input*, jika elemen adalah elemen dasar, maka kembalikan ElementNode berisi elemen tersebut.
 - ii. Untuk setiap resep yang memungkinkan dari elemen tersebut:
 1. Jika jumlah resep yang ada pada akar pohon sama atau lebih dari target, maka keluar dari *loop*.
 2. Panggil fungsi lagi terhadap bahan pertama resep dalam *thread* lain, dan terhadap bahan kedua resep dalam *thread* saat itu.
 - iii. Kembalikan ElementNode
4. Program mendapatkan 1 resep Atomic Bomb.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

A. Spesifikasi Teknis Program

Program dibuat dengan arsitektur aplikasi web pada umumnya yaitu *front-end*, *back-end*, dan sistem basis data. Next.js digunakan sebagai framework *front-end*, GO sebagai *back-end*, dan Postgres sebagai sistem basis data.



Bagian *back-end* dipisah menjadi beberapa file untuk meningkatkan modularitas program dan mempermudah proses implementasi.

a. Scraper

Kode untuk *web scraping* diimplementasi dalam file `scraper.go`. Pustaka atau *framework web scraping* yang digunakan adalah `gocolly`.

```
scraper.go

package scraper

import (
    "database/sql"
    "flab/internal/db"
    "fmt"
    "log"
    "strings"

    "github.com/gocolly/colly"
)

func getElementType(index int) int {
    switch index {
    case 1:
        return index - 1
    case 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17:
        return index - 2
    default:
        return -1
    }
}

func ScrapeElements(dbConn *sql.DB) error {
    db.ClearDB(dbConn)

    url :=
"https://little-alchemy.fandom.com/wiki/Elements_(Little_Alch
emy_2)"
    c :=
colly.NewCollector(colly.AllowedDomains("little-alchemy.fando
m.com"))

    tableIndex := 0
    elementCounter := 0
    recipeCounter := 0
    c.OnHTML("table.list-table", func(table
*colly.HTMLElement) {
        tableIndex++
        elementType := getElementType(tableIndex)
        if elementType == -1 {
            return
        }

        // each element generated
    })
}
```

```

        table.ForEach("tbody tr", func(_ int, h
*colly.HTMLElement) {
    element :=
strings.TrimSpace(h.ChildText("td:first-of-type a"))
    if element == "" || element == "Time" ||
element == "Ruins" || element == "Archeologist" {
        return
    }

    elementCounter++

    aTags := h.DOM.Find("td:nth-of-type(1) a")
    imgUrl, _ :=
aTags.Eq(0).Find("img").Attr("data-src")

    err := db.InsertElement(dbConn, element,
imgUrl, elementType)
    if err != nil {
        log.Printf("Error inserting element
'%s': %v", element, err)
        return
    }

    h.ForEach("td:nth-of-type(2) li", func(_
int, li *colly.HTMLElement) {
        recipeCounter++
        aTags := li.DOM.Find("a")

        if aTags.Length() < 2 {
            return
        }

        // imgUrl1, _ :=
aTags.Eq(0).Find("img").Attr("data-src")
        // imgUrl2, _ :=
aTags.Eq(2).Find("img").Attr("data-src")
        ingredient1 :=
strings.TrimSpace(aTags.Eq(1).Text())
        ingredient2 :=
strings.TrimSpace(aTags.Eq(3).Text())

        if ingredient1 == "Time" ||
ingredient2 == "Time" || ingredient1 == "Ruins" ||
ingredient2 == "Ruins" || ingredient1 == "Archeologist" ||
ingredient2 == "Archeologist" {
            return
        }

        // Insert into recipes table
        err := db.InsertRecipe(dbConn,
element, ingredient1, ingredient2)
        if err != nil {
            log.Printf("Error inserting

```

```

        recipe for element '%s': %v", element, err)
                return
            }

        }
    }

    c.OnRequest(func(r *colly.Request) {
        fmt.Println("Visiting ", r.URL)
    })

    c.OnError(func(r *colly.Response, e error) {
        fmt.Print(e.Error())
    })

    if err := c.Visit(url); err != nil {
        return fmt.Errorf("failed to visit URL: %w", err)
    }
    fmt.Printf("Scraping success: %d elements, %d
recipes\n", elementCounter, recipeCounter)
    return nil
}

```

Scraping bekerja dengan memproses elemen-elemen HTML pembentuk website yang ditarget. Untuk kasus Little Alchemy 2, data-data elemen termasuk nama, gambar, dan *tier* dipisah dalam tiap table, sehingga memudahkan *scraping*. Sedangkan data tiap resep didapatkan sebagai elemen-elemen list di tiap elemen.

b. Model

Paket Model berisi struktur-struktur data yang digunakan dalam program beserta metode-metode yang dimiliki struktur data tersebut.

element.go

```

package models

import "sync"

type ElementType struct {
    Name      string
    ImageUrl string
    Type      string
}

type ElementNode struct {
    Index          int           `json:"index"`
    Name           string        `json:"name"`
    Parent         *ElementNode `json:"-"`
}

```

```

        IsValid      bool           `json:"-"`
        ValidRecipeIdx []int        `json:"validRecipeIdx"`
        Recipes       []*RecipeNode `json:"recipes"`
    }

func (node *ElementNode) checkValidRecipe(recipeIdx int) {
    node.Name

    recipe := node.Recipes[recipeIdx]
    if recipe.Ingredient1 == nil
        || recipe.Ingredient2 == nil {
        return
    }

    if recipe.Ingredient1.IsValid
        && recipe.Ingredient2.IsValid {
        for len(node.ValidRecipeIdx) <= recipeIdx {
            node.ValidRecipeIdx =
                append(node.ValidRecipeIdx, 0)
        }

        node.ValidRecipeIdx[recipeIdx] =
            sumSlice(node.Recipes[recipeIdx].Ingredient1.ValidRecipeIdx)
        *
        sumSlice(node.Recipes[recipeIdx].Ingredient2.ValidRecipeIdx)
            node.setValid()
    }
}

func (node *ElementNode) setValid() {
    node.IsValid = true
    if node.Parent != nil {
        node.Parent.checkValidRecipe(node.Index)
    }
}

```

Suatu elemen direpresentasikan sebagai tipe data ElementType di dalam program. Sesuai dengan relasi elements di basis data, elemen memiliki atribut nama, (pranala) gambar, dan tipe/tier elemen tersebut. Namun untuk representasi elemen di dalam pohon resep, digunakan tipe ElementNode yang memiliki atribut penunjang struktur data pohon yaitu recipes sebagai larik resep-resep elemen tersebut.

recipe.go

```
package models
```

```
import "sync"
```

```

type RecipeType struct {
    Element      string
    Ingredient1 string
    Ingredient2 string
}

type RecipeNode struct {
    Ingredient1 *ElementNode `json:"ingredient1"`
    Ingredient2 *ElementNode `json:"ingredient2"`
}

type RecipeQueue struct {
    recipe [] *RecipeNode
    mu     sync.Mutex
}

func (queue *RecipeQueue) enqueue(recipe *RecipeNode) {
    queue.mu.Lock()
    defer queue.mu.Unlock()
    queue.recipe = append(queue.recipe, recipe) // Simply
append to enqueue.
    // fmt.Println("Enqueued:", *recipe)
}

func (queue *RecipeQueue) dequeue() *RecipeNode {
    queue.mu.Lock()
    defer queue.mu.Unlock()
    if len(queue.recipe) == 0 {
        return nil
    }
    element := (queue.recipe)[0] // The first element is
the one to be dequeued.
    queue.recipe = (queue.recipe)[1:]
    return element
}

func (queue *RecipeQueue) isEmpty() bool {
    queue.mu.Lock()
    defer queue.mu.Unlock()
    return len(queue.recipe) == 0
}

```

Seperti yang telah dijelaskan sebelumnya, `ElementNode` memiliki laris `RecipeNode`. Hal ini bertujuan untuk mempermudah *grouping* elemen atau bahan-bahan yang membentuk suatu resep elemen.

`tree.go`

```

func DFS(db *sql.DB, element string, targetCount int)
(*ElementNode, error) {

```

```

        ctx := context.Background()
        sem := make(chan struct{}, runtime.NumCPU())
        root, err := DFSRecursive(ctx, db, nil, element,
targetCount, sem)
        CutTree(root)
        return root, err
    }

func DFSRecursive(ctx context.Context, db *sql.DB, parentNode
*ElementNode, element string, targetCount int, sem chan
struct{}) (*ElementNode, error) {
    node := &ElementNode{Name: element, Parent: parentNode,
IsValid: false}

    if isBasicElement(element) {
        node.ValidRecipeIdx = append(node.ValidRecipeIdx,
1)
        node.setValid()
        return node, nil
    }

    typeQuery := "SELECT type FROM elements WHERE name =
\$1"
    row := db.QueryRowContext(ctx, typeQuery, element)
    var elementType int
    if err := row.Scan(&elementType); err != nil {
        return nil, err
    }

    query := "SELECT ingredient1, ingredient2 FROM recipes
WHERE element = \$1"
    rows, err := db.QueryContext(ctx, query, element)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    var recipes []Recipe
    for rows.Next() {
        var ing1, ing2 string
        if err := rows.Scan(&ing1, &ing2); err != nil {
            continue
        }

        var type1, type2 int
        row = db.QueryRowContext(ctx, typeQuery, ing1)
        if err := row.Scan(&type1); err != nil || type1
>= elementType {
            continue
        }
        row = db.QueryRowContext(ctx, typeQuery, ing2)
        if err := row.Scan(&type2); err != nil || type2
>= elementType {

```

```

        continue
    }

    recipes = append(recipes, Recipe{Ingredient1:
        ing1, Ingredient2: ing2})
    }

    if len(recipes) == 0 {
        return nil, fmt.Errorf("no valid recipe found for
        %s", element)
    }

    tmp := targetCount
    targetCount = int(math.Ceil(float64(targetCount) /
        float64(len(recipes)))))

    i := 0
==

tryAcquire := func() bool {
    // return true
    select {
    case sem <- struct{}{}:
        return true
    default:
        return false
    }
}

release := func() {
    <-sem
}

for _, recipe := range recipes {
    if i > 0 && tmp < 1 {
        break
    }

    recipeNode := &RecipeNode{}
    node.Recipes = append(node.Recipes, recipeNode)
    var err1, err2 error
    var wg sync.WaitGroup
    if ctx.Err() != nil {
        break
    }

    if tryAcquire() {
        wg.Add(1)
        go func() {
            defer release()
            defer wg.Done()
            recipeNode.Ingredient1, err1 =
                DFSRecursive(ctx, db, node, recipe.Ingredient1, targetCount,

```

```

sem)
        } ()
    } else {
        recipeNode.Ingredient1, err1 =
DFSRecursive(ctx, db, node, recipe.Ingredient1, targetCount,
sem)
    }

        recipeNode.Ingredient2, err2 = DFSRecursive(ctx,
db, node, recipe.Ingredient2, targetCount, sem)
        wg.Wait()

        hasValid := true
        if err1 != nil {
            hasValid = false
        }
        if err2 != nil {
            hasValid = false
        }

        if hasValid {
            recipeNode.Ingredient1.Index = i
            recipeNode.Ingredient2.Index = i

            recipeNode.Ingredient1.setValid()
            recipeNode.Ingredient2.setValid()

            i++
            x := sumSlice(node.ValidRecipeIdx)
            // fmt.Printf("SDFJSFH %s %d\n", element, x)
            tmp -= x
        }
    }

    if i == 0 {
        return node, fmt.Errorf("no valid sub-recipes for
%s", element)
    } else {
        return node, nil
    }
}

func BFSLive(db *sql.DB, element string, targetCount int,
emit func(*ElementNode)) (*ElementNode, error) {
    // initialize root node
    root := &ElementNode{Name: element, Parent: nil,
IsValid: false}

    // use queue for BFS
    queue := RecipeQueue{}
    queue.enqueue(&RecipeNode{Ingredient1: root,
Ingredient2: nil})
}

```

```

var wg sync.WaitGroup

for !queue.isEmpty() {
    time.Sleep(100 * time.Millisecond)

    currentRecipe := queue.dequeue()

    currentNode1 := currentRecipe.Ingredient1
    currentNode2 := currentRecipe.Ingredient2

    branchHitTarget := false
    nodeptr := currentNode1
    for nodeptr != nil {
        recipeCount :=
sumSlice(nodeptr.ValidRecipeIdx)
        if recipeCount >= targetCount {

            // early stop
            branchHitTarget = true
            break
        }
        nodeptr = nodeptr.Parent
    }
    if branchHitTarget {
        continue
    }

    // both basic element
    if currentNode1 != nil && currentNode2 != nil &&
isBasicElement(currentNode1.Name) &&
isBasicElement(currentNode2.Name) {
        currentNode1.ValidRecipeIdx =
append(currentNode1.ValidRecipeIdx, 1)
        currentNode2.ValidRecipeIdx =
append(currentNode2.ValidRecipeIdx, 1)
        currentNode1.setValid()
        currentNode2.setValid()
        emit(root)
        continue
    }

    wg.Add(2)
    go func(node *ElementNode) {
        defer wg.Done()
        if node == nil {
            return
        }
        processNodeBFS(db, node, &queue)
    }(currentNode1)

    go func(node *ElementNode) {
        defer wg.Done()

```

```

        if node == nil {
            return
        }
        processNodeBFS(db, node, &queue)
    } (currentNode2)

    wg.Wait()
    emit(root)
}

1
CutTree(root)
return root, nil
}

func BFS(db *sql.DB, element string, targetCount int) (*ElementNode, error) {
    // initialize root node
    root := &ElementNode{Name: element, Parent: nil,
    IsValid: false}

    // use queue for BFS
    queue := RecipeQueue{}
    queue.enqueue(&RecipeNode{Ingredient1: root,
    Ingredient2: nil})

    var wg sync.WaitGroup

    for !queue.isEmpty() {

        currentRecipe := queue.dequeue()

        currentNode1 := currentRecipe.Ingredient1
        currentNode2 := currentRecipe.Ingredient2

        branchHitTarget := false
        nodeptr := currentNode1
        for nodeptr != nil {
            recipeCount :=
sumSlice(nodeptr.ValidRecipeIdx)
            if recipeCount >= targetCount {
                fmt.Printf("%d %s VALID RECIPE IDX\n",
targetCount, nodeptr.Name)
                // early stop
                branchHitTarget = true
                break
            }
            nodeptr = nodeptr.Parent
        }
        if branchHitTarget {
            continue
        }
    }
}

```

```

        // both basic element
        if currentNode1 != nil && currentNode2 != nil &&
            isBasicElement(currentNode1.Name) &&
            isBasicElement(currentNode2.Name) {
            currentNode1.ValidRecipeIdx =
                append(currentNode1.ValidRecipeIdx, 1)
            currentNode2.ValidRecipeIdx =
                append(currentNode2.ValidRecipeIdx, 1)
            currentNode1.setValid()
            currentNode2.setValid()
            continue
        }

        wg.Add(2)
        go func(node *ElementNode) {
            defer wg.Done()
            if node == nil {
                return
            }
            processNodeBFS(db, node, &queue)
        }(currentNode1)

        go func(node *ElementNode) {
            defer wg.Done()
            if node == nil {
                return
            }
            processNodeBFS(db, node, &queue)
        }(currentNode2)

        wg.Wait()
    }

    // cut invalid subtrees
    CutTree(root)
    return root, nil
}

func processNodeBFS(db *sql.DB, node *ElementNode, queue
*RecipeQueue) {
    if node == nil {
        return
    }

    if isBasicElement(node.Name) {
        node.ValidRecipeIdx = append(node.ValidRecipeIdx,
1)
        node.setValid()
        return
    }

    typeQuery := "SELECT type FROM elements WHERE name =

```

```

$1"
    row := db.QueryRow(typeQuery, node.Name)
    var elementType int
    err := row.Scan(&elementType)
    if err == sql.ErrNoRows || err != nil {
        return
    }

    query := "SELECT ingredient1, ingredient2 FROM recipes
WHERE element = $1"
    rows, err := db.Query(query, node.Name)
    if err != nil {
        return
    }
    defer rows.Close()

    i := 0
    for rows.Next() {
        var ingredient1, ingredient2 string
        if err := rows.Scan(&ingredient1, &ingredient2);
err != nil {
            continue
        }

        // Do not continue path if recipes are higher
type
        query := "SELECT type FROM elements WHERE name =
$1"
        row := db.QueryRow(query, ingredient1)
        var elementType1 int
        err := row.Scan(&elementType1)
        if err == sql.ErrNoRows {
            continue
        } else if err != nil {
            return
        }

        if elementType1 >= elementType {
            continue
        }

        query = "SELECT type FROM elements WHERE name =
$1"
        row = db.QueryRow(query, ingredient2)
        var elementType2 int
        err = row.Scan(&elementType2)
        if err == sql.ErrNoRows {
            continue
        } else if err != nil {
            return
        }

        if elementType2 >= elementType {

```

```

        continue
    }

        child1 := &ElementNode{Name: ingredient1, Parent:
node, Index: i}
        child2 := &ElementNode{Name: ingredient2, Parent:
node, Index: i}

        newRecipe := &RecipeNode{Ingredient1: child1,
Ingredient2: child2}

        queue.enqueue(newRecipe)
        node.Recipes = append(node.Recipes, newRecipe)

        i++
    }
}

func CutTree(node *ElementNode) {
    if node == nil {
        return
    }

    var indicesToDelete []int
    // fmt.Printf("cutting %s\n", node.Name)
    for i, child := range node.Recipes {

        if !child.Ingredient1.IsValid ||
!child.Ingredient2.IsValid {
            indicesToDelete = append(indicesToDelete, i)
        }
    }

    for j := len(indicesToDelete) - 1; j >= 0; j-- {
        idx := indicesToDelete[j]
        node.Recipes = slices.Delete(node.Recipes, idx,
idx+1)
    }
    for _, child := range node.Recipes {
        CutTree(child.Ingredient1)
        CutTree(child.Ingredient2)
    }
}

func isBasicElement(element string) bool {
    return element == "Water" || element == "Air" ||
element == "Fire" || element == "Earth"
}

func sumSlice(numbers []int) int {
    sum := 0
    for _, num := range numbers {

```

```

        sum += num
    }
    return sum
}

```

tree.go berisi logika pencarian DFS dan BFS. Algoritma yang digunakan serupa dengan yang telah dijelaskan di bab sebelumnya. Implementasi *multithreading* dilakukan dengan memanfaatkan *goroutine*. Untuk membatasi penggunaan *goroutine*, digunakan juga *context* dan *semaphore*.

c. DB

Paket db berisi penghubung program dengan sistem basis data beserta pengurus *query-query*.

connection.go
<pre> package db import ("database/sql" "log" "os" _ "github.com/lib/pq") func ConnectDB() *sql.DB { db, err := sql.Open("postgres", os.Getenv("DATABASE_URL")) if err != nil { log.Fatal(err) } return db } func ClearDB(dbConn *sql.DB) { // create table if it doesn't exist _, err := dbConn.Exec("DROP TABLE IF EXISTS recipes") if err != nil { log.Fatal(err) } _, err = dbConn.Exec("DROP TABLE IF EXISTS elements") if err != nil { log.Fatal(err) } _, err = dbConn.Exec("CREATE TABLE IF NOT EXISTS </pre>

```

elements (name TEXT, image_url TEXT, type SMALLINT)")
    if err != nil {
        log.Fatal(err)
    }
_, err = dbConn.Exec("CREATE TABLE IF NOT EXISTS
recipes (element TEXT, ingredient1 TEXT, ingredient2 TEXT)")
    if err != nil {
        log.Fatal(err)
    }
}
}

```

queries.go

```

package db

import (
    "database/sql"
    "flab/internal/models"
)

func GetElements(db *sql.DB) ([]models.ElementType, error) {
    rows, err := db.Query("SELECT * FROM elements")
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    var elements []models.ElementType
    for rows.Next() {
        var element models.ElementType
        if err := rows.Scan(&element.Name,
&element.ImageUrl, &element.Type); err != nil {
            return nil, err
        }
        elements = append(elements, element)
    }

    return elements, rows.Err()
}

func GetElement(db *sql.DB, elementToQuery string)
(models.ElementType, error) {
    query := "SELECT * FROM elements WHERE name = $1"

    row := db.QueryRow(query, elementToQuery)

    var element models.ElementType
    err := row.Scan(&element.Name, &element.ImageUrl,
&element.Type)

    return element, err
}

```

```

func GetRecipes(db *sql.DB) ([]models.RecipeType, error) {
    rows, err := db.Query("SELECT * FROM recipes")
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    var recipes []models.RecipeType
    for rows.Next() {
        var recipe models.RecipeType
        if err := rows.Scan(&recipe.Element,
&recipe.Ingredient1, &recipe.Ingredient2); err != nil {
            return nil, err
        }
        recipes = append(recipes, recipe)
    }

    return recipes, rows.Err()
}

func InsertElement(db *sql.DB, element string, imgUrl string,
elementType int) error {
    sqlStatement := `

        INSERT INTO elements (name, image_url, type)
        VALUES ($1, $2, $3)`
    _, err := db.Exec(sqlStatement, element, imgUrl,
elementType)
    return err
}

func InsertRecipe(db *sql.DB, element string, ingredient1
string, ingredient2 string) error {
    sqlStatement := `

        INSERT INTO recipes (element, ingredient1, ingredient2)
        VALUES ($1, $2, $3)`
    _, err := db.Exec(sqlStatement, element, ingredient1,
ingredient2)
    return err
}

```

d. API

Paket API berisi rute-rute API yang diekspos ke *front-end*.

handlers.go

```
package api
```

```
import (
```

```

    "database/sql"
    "encoding/json"
    "fmt"
    "net/http"
    "strconv"
    "strings"

    "flab/internal/db"
    "flab/internal/models"

    "github.com/gorilla/mux"
)

func ScrapeElementsHandler(dbConn *sql.DB) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        err := scraper.ScrapeElements(dbConn)
        if err != nil {
            http.Error(w, err.Error(),
http.StatusInternalServerError)
            return
        }
        w.WriteHeader(http.StatusOK)
        w.Write([]byte("Scraping completed
successfully"))
    }
}

func GetElementsHandler(dbConn *sql.DB) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        elements, err := db.GetElements(dbConn)
        if err != nil {
            http.Error(w, err.Error(),
http.StatusInternalServerError)
            return
        }
        json.NewEncoder(w).Encode(elements)
    }
}

func GetRecipesHandler(dbConn *sql.DB) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        recipes, err := db.GetRecipes(dbConn)
        if err != nil {
            http.Error(w, err.Error(),
http.StatusInternalServerError)
            return
        }
        json.NewEncoder(w).Encode(recipes)
    }
}

func GetElementHandler(dbConn *sql.DB) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {

```

```

        vars := mux.Vars(r)
        elementToQuery := vars["element"]

        element, err := db.GetElement(dbConn,
elementToQuery)

        if err != nil {
            http.Error(w, createErrorResponse("Element
not found", http.StatusBadRequest), http.StatusBadRequest)
            return
        }

        json.NewEncoder(w).Encode(element)

    }

}

func GetRecipeHandler(db *sql.DB) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        element := r.URL.Query().Get("element")
        strategy :=
strings.ToLower(r.URL.Query().Get("strategy"))
        count, err :=
strconv.Atoi(r.URL.Query().Get("count"))

        if element == "" {
            http.Error(w, createErrorResponse("Element
parameter required", http.StatusBadRequest),
http.StatusBadRequest)
            return
        }

        if err != nil {
            http.Error(w, createErrorResponse("Target
count parameter error", http.StatusBadRequest),
http.StatusBadRequest)
            return
        }

        if count <= 0 {
            http.Error(w, createErrorResponse("Target
count must be positive integer", http.StatusBadRequest),
http.StatusBadRequest)
            return
        }

        if strategy != "bfs" && strategy != "dfs" {
            http.Error(w, createErrorResponse("Strategy
parameter error", http.StatusBadRequest),
http.StatusBadRequest)
            return
        }
    }
}

```

```

        }

        if strategy == "dfs" {
            fmt.Println("DFS")

            root, err := models.DFS(db, element, count)
            if err != nil {
                http.Error(w,
createErrorResponse(err.Error(),
http.StatusInternalServerError),
http.StatusInternalServerError)
                    return
            }

            w.Header().Set("Content-Type",
"application/json")
                json.NewEncoder(w).Encode(root)

        } else {
            fmt.Println("BFS")

            root, err := models.BFS(db, element, count)
            if err != nil {
                http.Error(w,
createErrorResponse(err.Error(),
http.StatusInternalServerError),
http.StatusInternalServerError)
                    return
            }

            w.Header().Set("Content-Type",
"application/json")
                json.NewEncoder(w).Encode(root)
        }
        fmt.Println("done")

    }
}

func GetLiveRecipeHandler(db *sql.DB) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // Set SSE headers
        w.Header().Set("Content-Type",
"text/event-stream")
        w.Header().Set("Cache-Control", "no-cache")
        w.Header().Set("Connection", "keep-alive")

        flusher, ok := w.(http.Flusher)
        if !ok {
            http.Error(w, "Streaming unsupported",
http.StatusInternalServerError)
                    return
        }
    }
}

```

```

        element := r.URL.Query().Get("element")
        strategy :=
strings.ToLower(r.URL.Query().Get("strategy"))
        count, err :=
strconv.Atoi(r.URL.Query().Get("count"))

        if element == "" {
            http.Error(w, createErrorResponse("Element
parameter required", http.StatusBadRequest),
http.StatusBadRequest)
            return
        }

        if err != nil {
            http.Error(w, createErrorResponse("Target
count parameter error", http.StatusBadRequest),
http.StatusBadRequest)
            return
        }

        if count <= 0 {
            http.Error(w, createErrorResponse("Target
count must be positive integer", http.StatusBadRequest),
http.StatusBadRequest)
            return
        }

        if strategy != "bfs" && strategy != "dfs" {
            http.Error(w, createErrorResponse("Strategy
parameter error", http.StatusBadRequest),
http.StatusBadRequest)
            return
        }

        emit := func(node *models.ElementNode) {
            data, _ := json.Marshal(node)
            fmt.Fprintf(w, "data: %s\n\n", data)
            flusher.Flush()
        }

        if strategy == "dfs" {
            out, _ := models.DFSLive(db, element, count,
emit)
            data, _ := json.Marshal(out)
            fmt.Fprintf(w, "event: done\ndata: %s\n\n",
data)
            flusher.Flush()
        } else {
            out, _ := models.BFSLive(db, element, count,
emit)
            data, _ := json.Marshal(out)
            fmt.Fprintf(w, "event: done\ndata: %s\n\n",

```

```

data)
    flusher.Flush()
}
}

func createErrorResponse(message string, statusCode int) string {
    errorResponse := map[string]interface{}{
        "error":      true,
        "message":   message,
        "statusCode": statusCode,
    }
    errorJSON, _ := json.Marshal(errorResponse)
    return string(errorJSON)
}

```

middleware.go

```

package api

import "net/http"

func EnableCORS(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Access-Control-Allow-Origin",
        "*")
        w.Header().Set("Access-Control-Allow-Methods",
        "GET, POST, PUT, DELETE, OPTIONS")
        w.Header().Set("Access-Control-Allow-Headers",
        "Content-Type, Authorization")

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

        next.ServeHTTP(w, r)
    })
}

func JSONContentTypeMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type",
        "application/json")
        next.ServeHTTP(w, r)
    })
}

```

router.go

```
package api

import (
    "database/sql"

    "github.com/gorilla/mux"
)

func NewRouter(dbConn *sql.DB) *mux.Router {
    router := mux.NewRouter()
    router.HandleFunc("/api/go/scrape",
        ScrapeElementsHandler(dbConn)).Methods("GET")

    router.HandleFunc("/api/go/elements",
        GetElementsHandler(dbConn)).Methods("GET")
    router.HandleFunc("/api/go/recipes",
        GetRecipesHandler(dbConn)).Methods("GET")

    router.HandleFunc("/api/go/element/{element}",
        GetElementHandler(dbConn)).Methods("GET")
    router.HandleFunc("/api/go/recipe",
        GetRecipeHandler(dbConn)).Methods("GET")
    router.HandleFunc("/api/go/liverecipe",
        GetLiveRecipeHandler(dbConn)).Methods("GET")
    return router
}
```

Bagian *front-end* dipisah berdasarkan halaman-halaman yang ada serta komponen-komponen yang membentuk halaman.

a. Search Page

Fitur utama program ini dapat diakses di laman search, dimana terdapat antarmuka untuk memasukkan input dan menunjukkan pohon resep yang dihasilkan program. Di bagian ini *frontend* juga melakukan koneksi ke *backend* dengan melakukan pemanggilan API *backend*.

Untuk fitur *Live Update*, data *real-time* dikirim menggunakan Server-Sent Event (SSE). SSE memungkinkan untuk server mengirim data secara terus menerus selama koneksi masih dipertahankan.

search/page.tsx

```

export default function Page() {
    const router = useRouter();
    const context = useContext(DarkModeContext);

    if (!context) {
        throw new Error('No Context!');
    }

    const { darkMode } = context;
    const [elements, setElements] = useState([])
    useEffect(() => {
        axios.get('http://localhost:8000/api/go/elements')
            .then(res => setElements(res.data))
            .catch(err => console.log(err))
    }, [])

    // console.log(elements)
    const [selectedElement, setSelectedElement] = useState('');
    const [strategy, setStrategy] = useState('dfs');

    const [showNumberInput, setShowNumberInput] =
    useState(false);
    const [recipeCount, setRecipeCount] = useState(1);

    const [recipeTree, setRecipeTree] = useState<RecipeNodeType | null>(null);

    const [loading, setLoading] = useState(false);
    const [loadTime, setLoadTime] = useState<number | null>(null);

    const [useLiveSearch, setUseLiveSearch] = useState(false);

    const [nodeCount, setNodeCount] = useState();
    const [recipeFound, setRecipeFound] = useState();

    const handleToggleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
        setShowNumberInput(e.target.checked);
        if (!e.target.checked) {
            setRecipeCount(1);
        }
    };

    const handleSearch = () => {
        if (!selectedElement) {
            alert("Please select an element.");
            return;
        }

        setLoading(true);
        setLoadTime(null);
        const start = performance.now();

```

```

        const url =
`http://localhost:8000/api/go/recipe?element=${encodeURIComponent(selectedElement)}&strategy=${strategy}&count=${recipeCount}`;
        axios.get(url)
            .then(res => {
                const end = performance.now();
                setLoadTime(end - start);
                console.log(res.data);
                setRecipeTree(res.data.tree);
                setNodeCount(res.data.nodeCount);
                setRecipeFound(res.data.recipeCount);
            })
            .catch(err => {
                let errorMessage = "Error fetching recipe.";

                if (err.response && err.response.data) {
                    errorMessage = err.response.data.message || errorMessage;
                }

                console.error("Error fetching recipe:", errorMessage);
                alert(errorMessage);
            })
            .finally(() => setLoading(false));
    };

    const handleSSEMessage = (data: string) => {
        try {
            const parsed = JSON.parse(data);
            setRecipeTree(parsed);
            console.log(parsed);
        } catch (error) {
            console.error("Invalid SSE data:", error);
        }
    };
}

const handleLiveSearch = () => {
    if (!selectedElement) {
        alert("Please select an element.");
        return;
    }

    setLoading(true);
    setLoadTime(null);
    const start = performance.now();

    const url =
`http://localhost:8000/api/go/liverecipe?element=${encodeURIComponent(selectedElement)}&strategy=${strategy}&count=${recipeCount}`;

```

```

const eventSource = new EventSource(url);

eventSource.onmessage = (event) => {
    handleSSEMessage(event.data);
};

eventSource.addEventListener("done", (event) => {
    const parsed = JSON.parse(event.data);
    const end = performance.now();
    setLoadTime(end - start);
    setLoading(false);
    setRecipeFound(parsed.recipeCount);
    setNodeCount(parsed.nodeCount);
    eventSource.close();
});

eventSource.onerror = () => {
    console.error("SSE connection error.");
    eventSource.close();
    setLoading(false);
};

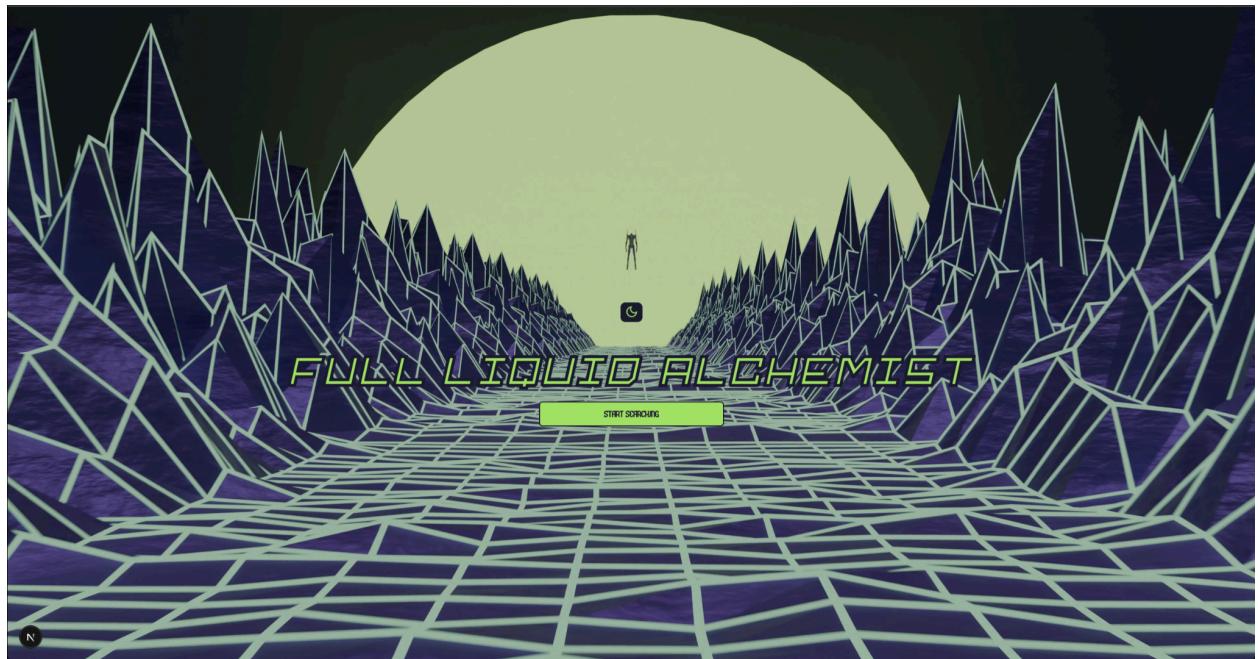
return (
    ... html content
);
}

```

B. Penjelasan Tata Cara Penggunaan Program

Pengguna bisa mengakses aplikasi dengan menjalankan aplikasi web secara lokal. Langkah umum untuk membuka dan menjalankan web secara lokal adalah sebagai berikut:

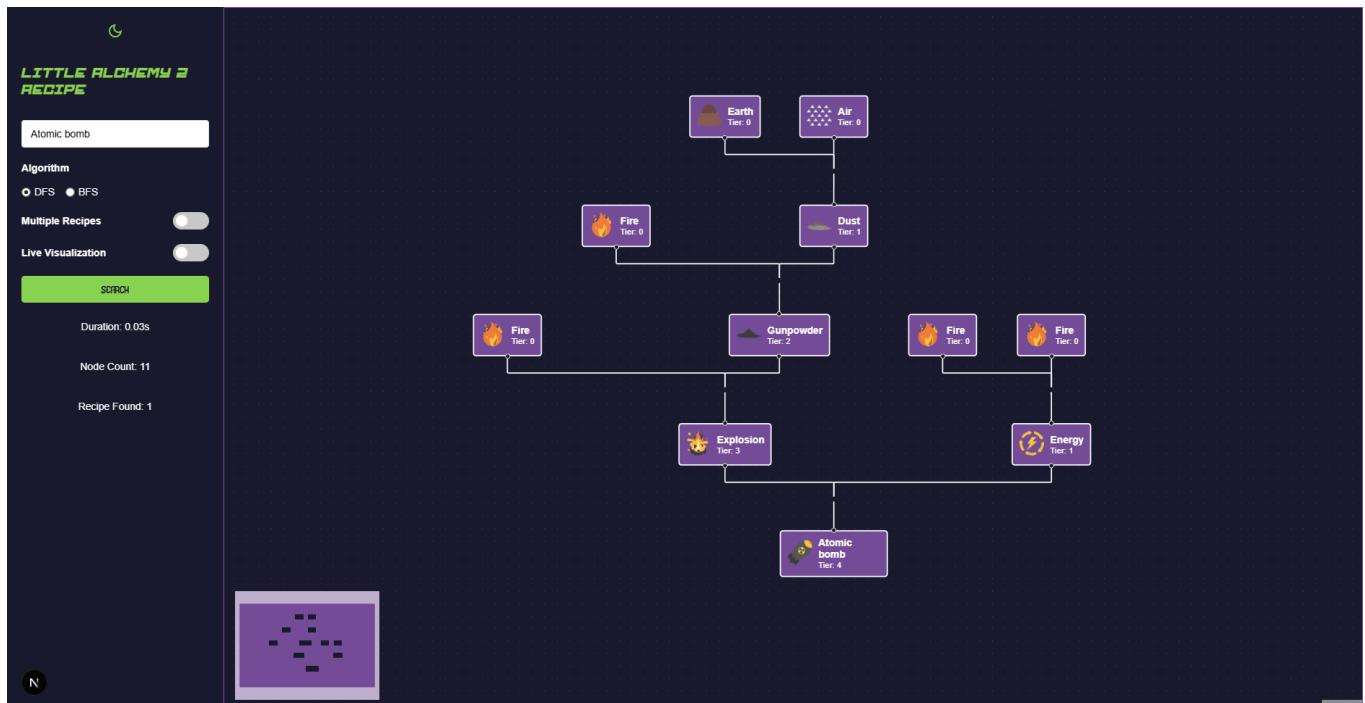
1. Lakukan docker compose build
2. Lakukan docker compose up
3. Buka <http://localhost:3000> (atau port terbuka selanjutnya jika sudah terisi)



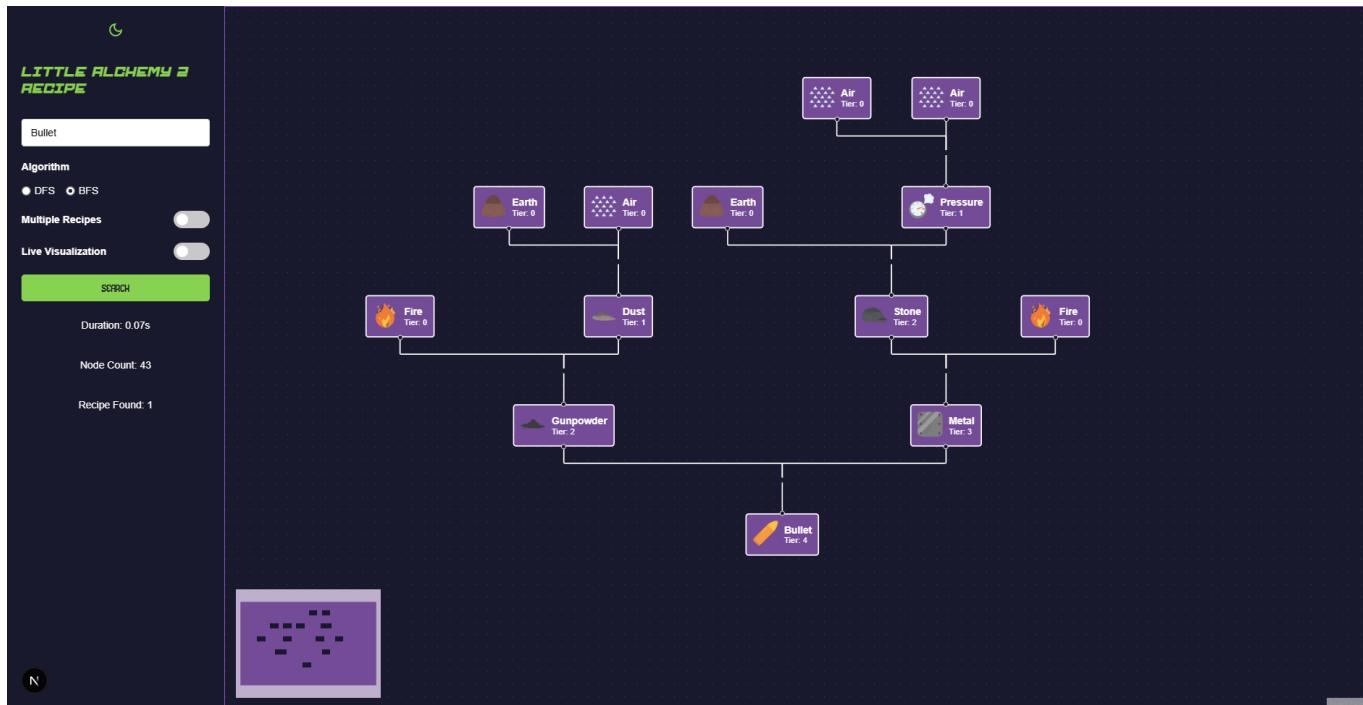
Gambar 11. Tampilan antarmuka halaman utama.

(sumber: Arsip kelompok)

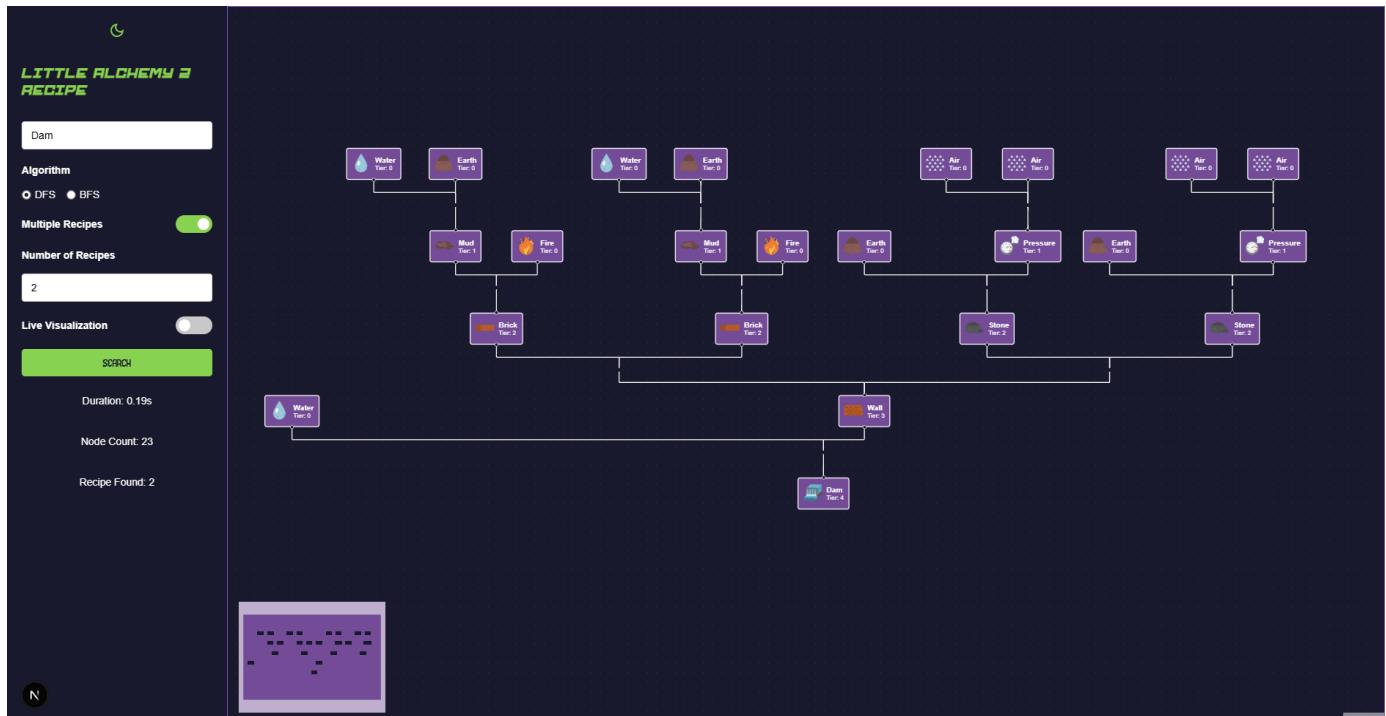
C. Hasil Pengujian



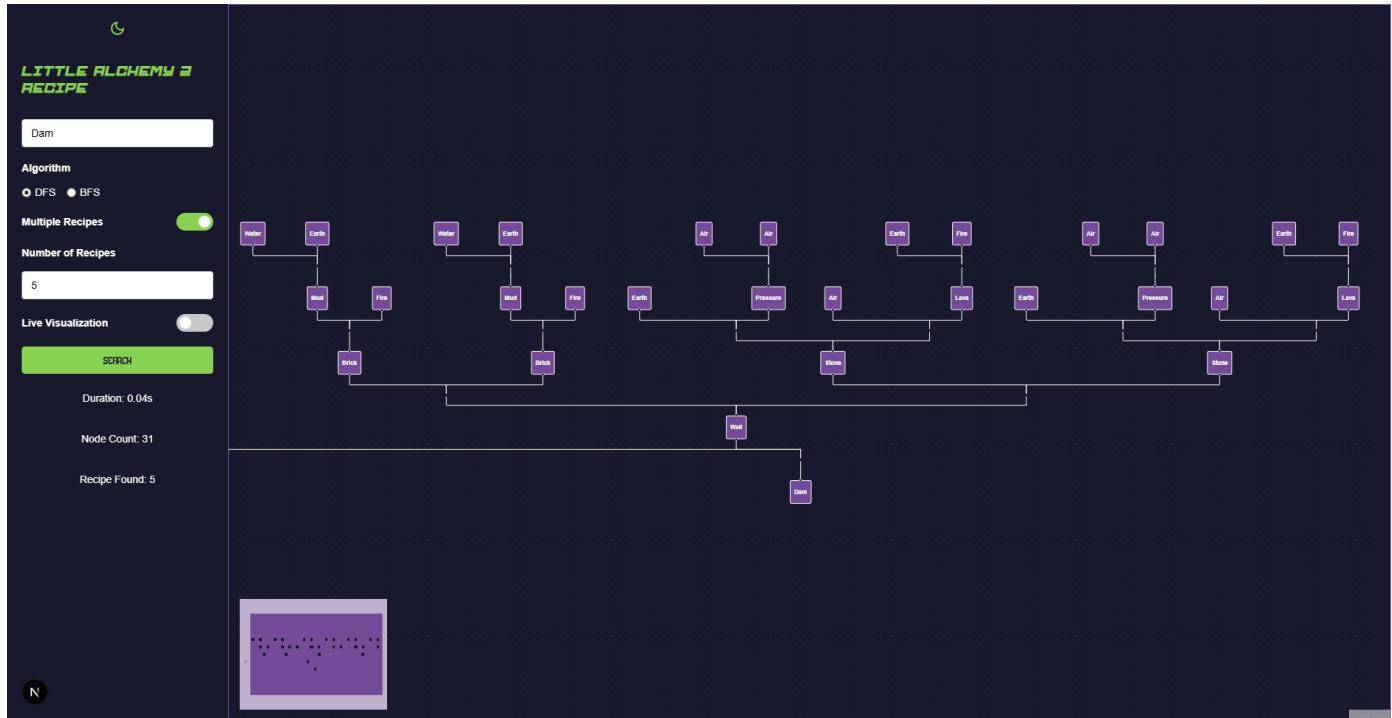
Gambar 12. Pengujian algoritma algoritma DFS dalam pencarian resep 'Atomic Bomb' (Single Recipe)



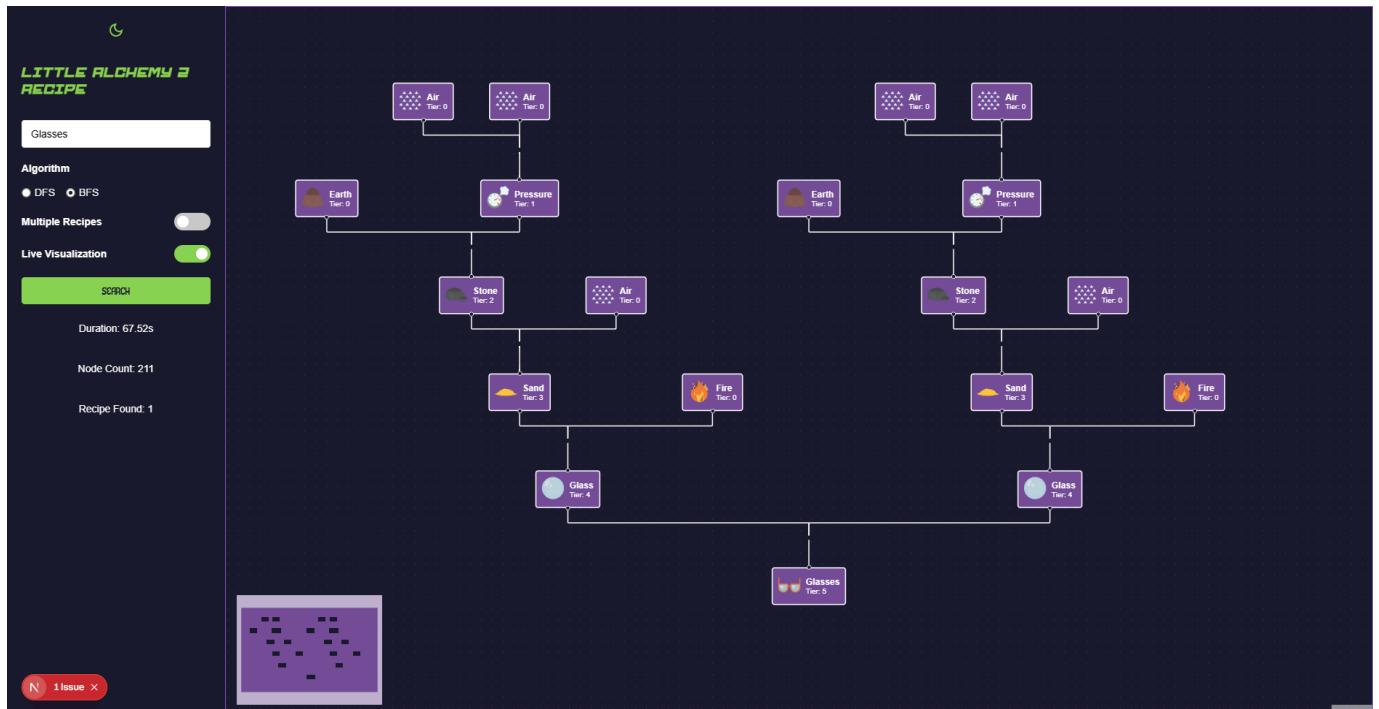
Gambar 13. Pengujian algoritma BFS dalam mencari resep ‘Bullet’ (Single Recipe)



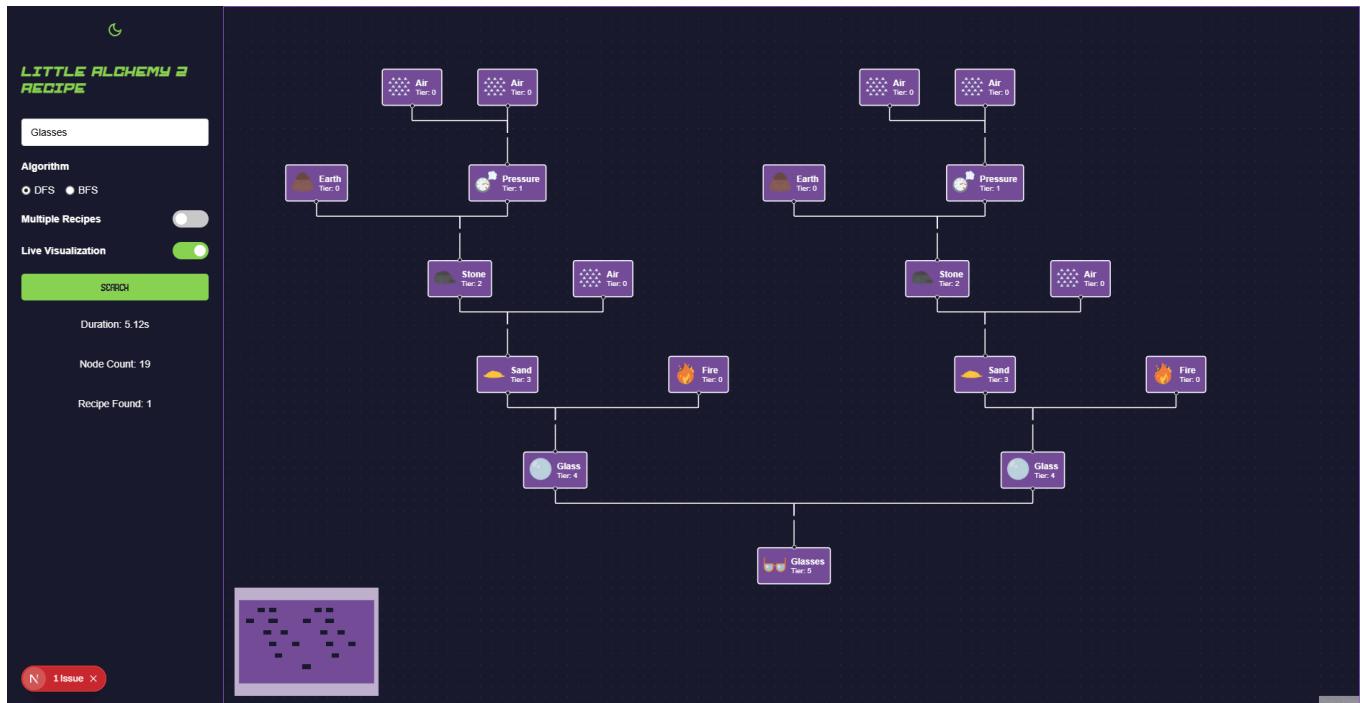
Gambar 14. Pengujian algoritma pencarian dalam mencari 2 resep untuk ‘Dam’



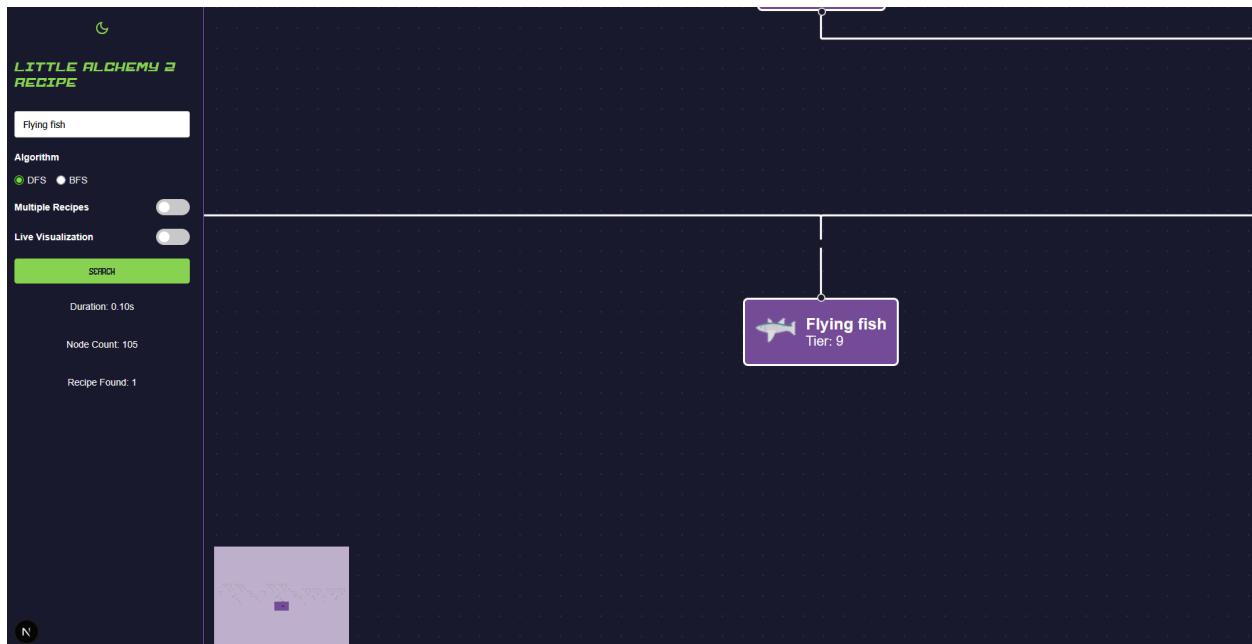
Gambar 15. Pengujian algoritma pencarian dalam mencari 5 resep untuk 'Dam'



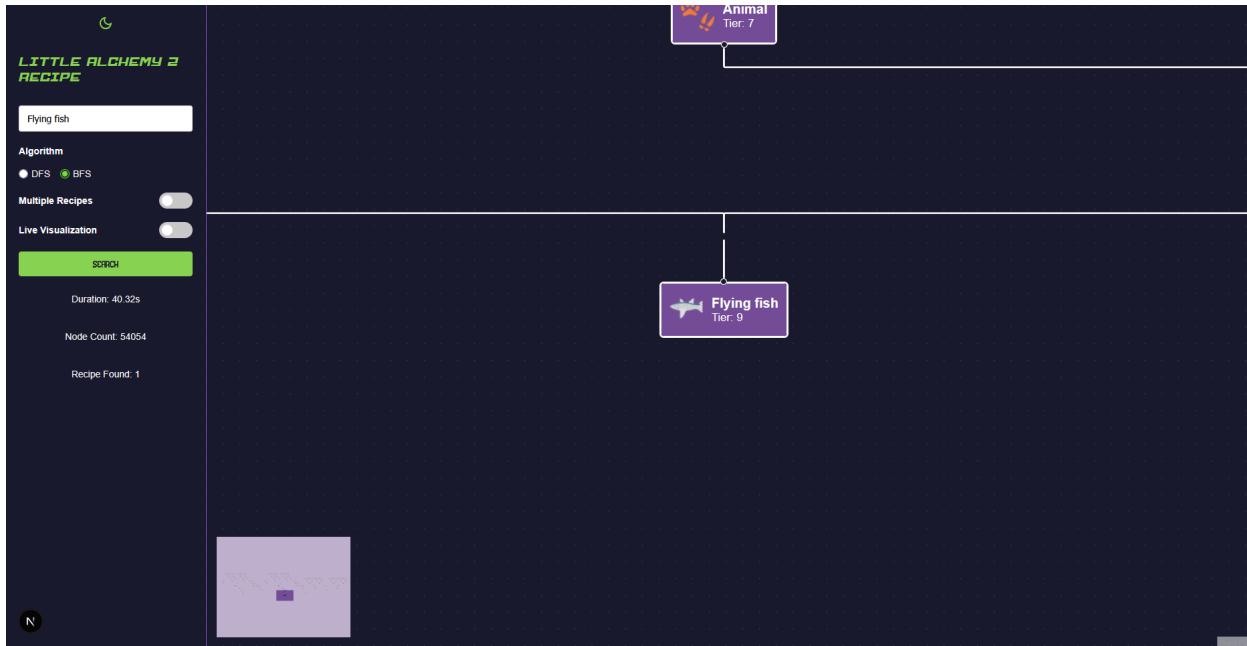
Gambar 16. Pengujian pencarian menggunakan BFS dan Live Visualization untuk mencari 'Glasses'



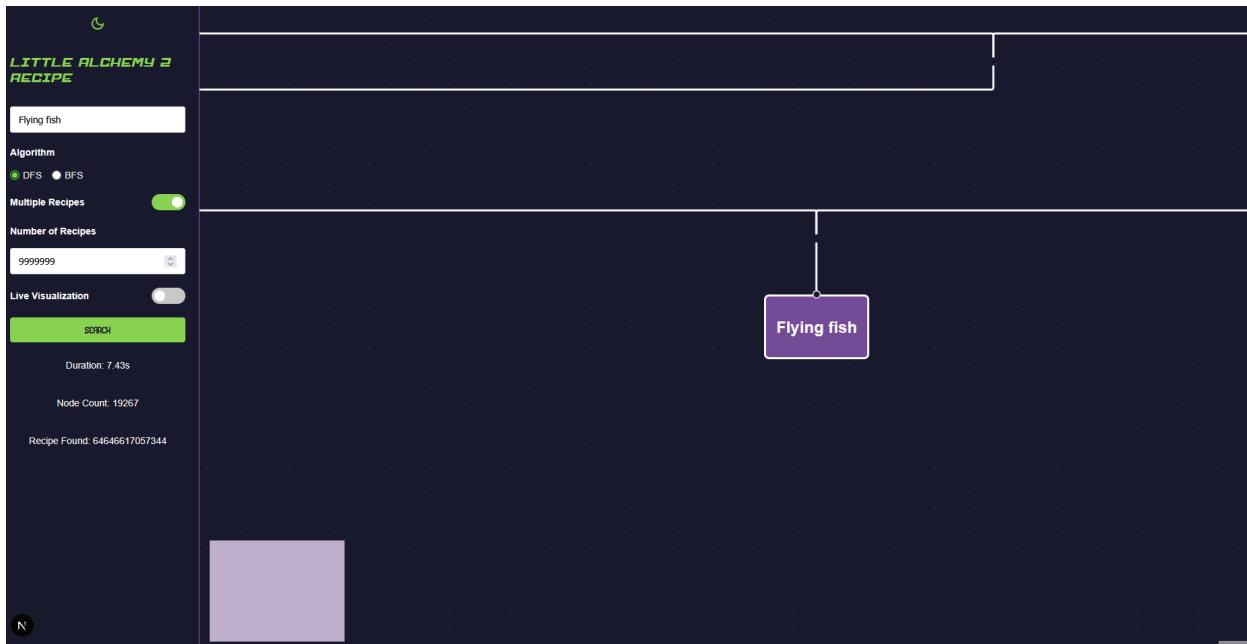
Gambar 17. Pengujian pencarian menggunakan DFS dan *Live Visualization* untuk mencari ‘Glasses’



Gambar 18. Pengujian pencarian menggunakan DFS pada elemen tier tinggi ‘Flying fish’



Gambar 19. Pengujian pencarian menggunakan BFS pada elemen tier tinggi 'Flying fish'



Gambar 20. Stress test menggunakan DFS pada elemen tier tinggi 'Flying fish'

D. Analisis Hasil

Berdasarkan waktu yang dibutuhkan dalam proses pencarian, didapatkan bahwa DFS relatif membutuhkan waktu yang lebih singkat dibandingkan BFS. Hal ini terutama terlihat saat opsi 'Live Visualization' diaktifkan (seperti yang ditampilkan pada Gambar 16 dan 17), di mana perbedaan durasi antara kedua algoritma menjadi lebih mencolok dan ketika kita mencari elemen tier tinggi *single recipe* sangat terlihat perbedaannya, DFS

hanya butuh 0.1 detik dan 105 Node Count, sedangkan BFS sampai 40 detik dan 54054 Node Count. Penyebab utamanya adalah karena BFS mengeksplorasi simpul-simpul secara lebih luas di setiap level, sehingga jumlah node yang harus diakses dan divisualisasikan lebih banyak dibandingkan DFS, yang fokus menyusuri satu jalur hingga kedalaman maksimal terlebih dahulu.

Secara internal, BFS memerlukan lebih banyak antrian dan pencatatan status kunjungan pada setiap simpul tetangga di level-level awal graf, sementara DFS bisa langsung "menyelam" ke dalam cabang yang mungkin saja mengarah pada solusi tanpa harus memeriksa seluruh simpul yang sejajar. Inilah yang menjadikan DFS lebih ringan secara waktu eksekusi, terutama saat visualisasi dijalankan secara langsung dan sinkron.

Namun, dari sisi akurasi hasil saat mencari multiple recipe, hasil pengujian menunjukkan bahwa BFS cenderung lebih konsisten dalam menghasilkan jumlah resep sesuai target. Hal ini terjadi karena BFS menelusuri semua kemungkinan kombinasi dari level terbawah ke atas secara merata, sehingga lebih mudah mengontrol berapa banyak solusi yang dihasilkan secara paralel. Sebaliknya, DFS cenderung mengejar solusi lebih cepat dalam satu cabang terlebih dahulu, yang bisa menyebabkan jumlah hasil yang ditemukan kurang merata, atau bahkan berhenti terlalu dini jika tidak dikontrol secara eksplisit.

Dengan demikian, terdapat trade-off antara kecepatan dan keluasan eksplorasi: DFS lebih cepat namun bisa melewatkkan solusi alternatif, sedangkan BFS lebih komprehensif namun lebih berat secara waktu dan memori, terutama pada graf yang luas pada elemen dengan *tier* tinggi.

BAB V

KESIMPULAN, SARAN, DAN REFLEKSI

A. Kesimpulan

Kesimpulan dari pengujian algoritma Breadth-First Search (BFS) dan Depth-First Search (DFS) dalam sistem pencarian resep My Little Alchemist adalah sebagai berikut:

1. DFS terbukti lebih cepat dalam melakukan pencarian dibandingkan BFS, terutama ketika. Hal ini disebabkan oleh strategi traversal DFS yang fokus pada satu jalur secara mendalam, sehingga jumlah simpul yang harus diproses dan divisualisasikan menjadi lebih sedikit.
2. Sebaliknya, BFS memiliki keunggulan dalam akurasi pencarian, khususnya saat mencari multiple recipe, karena kemampuannya menjelajahi semua kemungkinan kombinasi secara menyeluruh pada setiap level. Perbedaan ini menunjukkan bahwa masing-masing algoritma memiliki kelebihan yang berbeda: DFS lebih unggul dari sisi efisiensi waktu, sementara BFS lebih kuat dalam kelengkapan hasil.

Dari temuan tersebut, dapat disimpulkan bahwa tidak ada satu algoritma yang selalu optimal di segala kondisi. Pemilihan algoritma pencarian perlu mempertimbangkan kebutuhan pengguna, apakah lebih mengutamakan kecepatan atau kelengkapan solusi.

B. Saran

Berikut beberapa saran yang dapat diterapkan untuk meningkatkan efektivitas sistem pencarian resep di My Little Alchemist:

1. Integrasi pendekatan adaptif yang secara dinamis memilih antara DFS dan BFS berdasarkan parameter seperti target pencarian (single vs multiple recipe) atau ukuran graf.
2. Optimasi visualisasi untuk mengurangi beban saat BFS digunakan dalam mode Live Visualization, misalnya dengan hanya menampilkan simpul aktif atau mereduksi frame rate.
3. Eksperimen dengan varian algoritma lain, seperti Iterative Deepening DFS atau Limited BFS, yang menggabungkan keunggulan kedua metode.

4. Pengujian lebih lanjut pada graf skala besar dan skenario kompleks untuk mengidentifikasi batas performa masing-masing algoritma secara lebih akurat.

C. Refleksi

Selama proses pengembangan sistem pencarian resep pada proyek My Little Alchemist 2, kami mendapatkan banyak pemahaman baru, terutama terkait pemodelan graf non-konvensional dan bagaimana struktur data harus dirancang secara fleksibel sesuai dengan kebutuhan spesifik dari permasalahan. Tidak semua relasi dalam sistem dapat diwakili oleh model graf standar, dan hal ini menjadi tantangan menarik yang mendorong penulis untuk lebih kreatif dalam menyusun representasi data.

Pemilihan algoritma pencarian juga memberikan wawasan penting mengenai trade-off antara kecepatan dan kelengkapan hasil. Meskipun sebelumnya penulis menganggap DFS dan BFS hanya berbeda pada cara traversal, ternyata penerapannya dalam sistem nyata menghasilkan perbedaan yang signifikan terhadap performa dan akurasi hasil. Hal ini mengajarkan bahwa konteks penggunaan algoritma sangat mempengaruhi efektivitasnya.

Selain aspek teknis, proyek ini juga memperkuat keterampilan dalam hal pengujian, debugging, dan evaluasi algoritma, terutama saat menghadapi ketidaksesuaian antara hasil teori dan implementasi awal. Penulis menyadari pentingnya iterasi dan validasi berulang dalam membangun sistem yang akurat dan cepat.

LAMPIRAN

Tabel Implementasi Program:

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.		✓

Pembagian Tugas :

NIM	Nama	Tugas
13523123	Rhio Bimo Prakoso S	Laporan, Scrapping, Front-End, Back-End
13523152	M Kinan Arkansyaddad	Laporan, Scrapping, Front-End, Back-End
13523162	Fachriza Ahmad Setiyono	Laporan, Scrapping, Front-End, Back-End, Full-Stack

Link Github Repository :

https://github.com/L4mbads/Tubes2_FullLiquidAlchemistBrotherhood

Link Video :

<https://youtu.be/VvDhgct-2e0?si=6gzZ9w-jBqL85SFb>

Please decode!! : UEFjZjU1djZ6cVE=

Free Wallpaper (from us!) :  Hasil bekas dari video dan kebetulan bagus dan karena kami tidak suka gate-keeping, dibuatlah folder ini. Drive ini akan terus di-update jika ada tugas mendatang dan/atau kalau aku gabut :v. Stay-tuned :p

DAFTAR PUSTAKA

Munir, Rinaldi. 2025. BFS-DFS-Bag1.
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf>
. Diakses pada 1 Maret 2025.