

**LAPORAN TUGAS BESAR 2**  
**IF2211 STRATEGI ALGORITMA**

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



Disusun oleh :

Dzaky Aurelia Fawwaz	13523065
Carlo Angkisan	13523091
Barru Adi Utomo	13523101

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

**2025**

## DAFTAR ISI

<b>DAFTAR ISI.....</b>	<b>2</b>
<b>BAB I.....</b>	<b>3</b>
<b>BAB II.....</b>	<b>5</b>
2.1 Graf dan Penjelajahan Graf.....	5
2.2 Breadth First Search (BFS).....	5
2.3 Depth First Search (DFS).....	6
2.4 Bidirectional Search.....	7
2.5 Aplikasi Web Yang Dibangun.....	7
<b>BAB III.....</b>	<b>9</b>
3.1 Langkah-Langkah Pemecahan Masalah.....	9
3.2 Pemetaan Masalah Menjadi Elemen-Elemen Algoritma DFS dan BFS.....	9
3.3 Fitur Fungsional dan Arsitektur Aplikasi Web.....	10
3.4 Contoh Ilustrasi Kasus.....	15
<b>BAB IV.....</b>	<b>21</b>
4.1 Spesifikasi Teknis Program.....	21
4.2 Tata Cara Penggunaan Program.....	29
4.3 Hasil Pengujian.....	34
4.4 Analisis Hasil Pengujian.....	41
<b>BAB V.....</b>	<b>43</b>
5.1 Kesimpulan.....	43
5.2 Saran.....	43
5.3 Refleksi.....	43
<b>LAMPIRAN.....</b>	<b>45</b>
Tautan Repository Github.....	45
Tautan Video.....	45
Tabel Kelengkapan Spesifikasi.....	45
<b>DAFTAR PUSTAKA.....</b>	<b>46</b>

## 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

#### 2.1 Graf dan Penjelajahan Graf

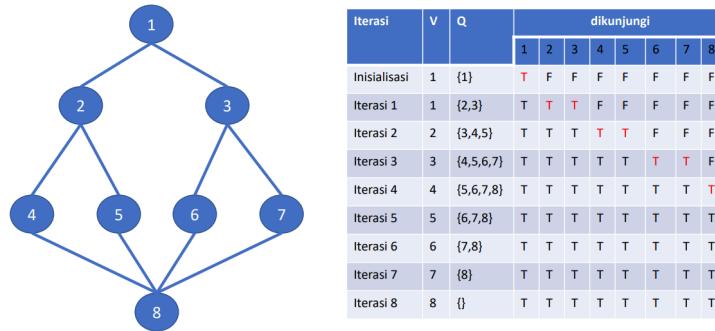
Graf merupakan salah satu struktur data fundamental yang digunakan untuk merepresentasikan hubungan antar objek melalui himpunan simpul yang dihubungkan oleh sisi. Berdasarkan arah hubungan antar simpul, graf dapat dibedakan menjadi graf terarah yang memiliki arah tertentu pada setiap sisi, dan graf tak terarah yang hubungan antar simpulnya bersifat dua arah. Penjelajahan graf atau *graph traversal* adalah proses sistematis untuk mengunjungi simpul-simpul dalam graf, baik untuk mencari informasi tertentu, memproses data, maupun untuk keperluan analisis lebih lanjut. Dalam pencarian solusi berbasis graf, terdapat dua pendekatan utama yang dapat digunakan. Pendekatan pertama adalah algoritma pencarian tanpa informasi atau *uninformed search*, yang tidak menggunakan informasi tambahan tentang posisi tujuan, melainkan hanya mengandalkan struktur graf yang ada. Contoh dari algoritma jenis ini antara lain *Breadth First Search*, *Depth First Search*, *Depth Limited Search*, *Iterative Deepening Search*, dan *Uniform Cost Search*. Pendekatan kedua adalah algoritma pencarian dengan informasi atau *informed search*, yang memanfaatkan heuristik untuk memperkirakan jalur yang lebih menjanjikan menuju tujuan, sehingga proses pencarian menjadi lebih efisien. Algoritma seperti *Best First Search* dan *A Star (A\*)* termasuk dalam kategori ini. Selain pendekatan pencarian, dalam proses membangun solusi menggunakan graf, terdapat pula perbedaan dalam cara representasi graf yang digunakan. Representasi graf dapat bersifat statis, yaitu graf sudah tersedia secara lengkap sebelum pencarian dimulai dan biasanya disimpan dalam bentuk struktur data seperti matriks ketetanggaan atau daftar ketetanggaan. Sebaliknya, graf juga bisa bersifat dinamis, di mana graf dibentuk seiring berjalannya proses pencarian, sehingga simpul dan sisi baru dapat ditambahkan sesuai kebutuhan pencarian. Pemilihan antara graf statis atau dinamis ini bergantung pada karakteristik masalah yang dihadapi dan berpengaruh terhadap efektivitas serta efisiensi proses pencarian solusi.

#### 2.2 Breadth First Search (BFS)

*Breadth First Search* (BFS) adalah algoritma penjelajahan graf yang bekerja dengan prinsip menjelajah semua simpul pada satu tingkat sebelum melanjutkan ke tingkat berikutnya. BFS dimulai dari simpul awal, kemudian mengunjungi semua simpul tetangga yang langsung terhubung, dilanjutkan dengan simpul-simpul yang terhubung ke tetangga tersebut, dan seterusnya, secara berurutan berdasarkan tingkat kedalaman dari simpul awal. Untuk mengelola urutan simpul yang akan dikunjungi, BFS menggunakan struktur data antrian (*queue*) sehingga simpul yang pertama kali ditemukan akan diproses terlebih dahulu. Salah satu keunggulan utama BFS adalah kemampuannya menemukan jalur terpendek dalam graf tidak berbobot, karena algoritma ini menjelajah level demi level. Kompleksitas waktu dari BFS adalah  $O(V + E)$ , di mana V adalah jumlah simpul

dan E adalah jumlah sisi, sehingga sangat efisien untuk graf besar dan padat. BFS banyak diterapkan dalam berbagai bidang seperti pencarian dalam labirin, perhitungan jarak terpendek, serta dalam algoritma jaringan seperti pencarian teman terdekat dalam jejaring sosial.

### BFS: Ilustrasi



14

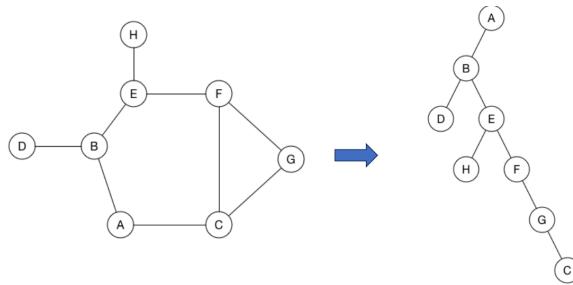
Gambar 3. Ilustrasi BFS

(Sumber:

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-\(2025\)-Bahan1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bahan1.pdf)

### 2.3 Depth First Search (DFS)

*Depth First Search* (DFS) adalah algoritma penjelajahan graf yang berfokus pada eksplorasi jalur sepanjang mungkin sebelum melakukan *backtrack* ke simpul sebelumnya untuk mengeksplorasi jalur lain. DFS mulai dari simpul awal, kemudian bergerak ke salah satu tetangganya, melanjutkan lagi ke tetangga dari simpul tersebut, dan seterusnya hingga mencapai simpul yang tidak memiliki tetangga yang belum dikunjungi. Jika tidak ada simpul lagi untuk dieksplorasi, algoritma akan mundur ke simpul sebelumnya dan melanjutkan eksplorasi dari sana. Implementasi DFS dapat dilakukan secara rekursif maupun menggunakan struktur data *stack*. DFS sangat berguna dalam menemukan komponen terhubung dalam graf, mendeteksi siklus dalam graf terarah maupun tidak terarah, melakukan topological sorting pada graf terarah, dan menyelesaikan berbagai jenis puzzle yang memerlukan eksplorasi jalur, seperti labirin. Dengan kompleksitas waktu sebesar  $O(V + E)$ , DFS juga sangat efisien dan banyak digunakan dalam pengembangan algoritma pencarian dan pengoptimalan. Selain itu, DFS memberikan dasar yang kuat untuk memahami struktur graf lebih dalam, seperti pohon pencarian dan algoritma untuk pengolahan graf berat



Urutan simpul-simpul yang dikunjungi secara DFS dari A → A, B, D, E, H, F, G, C

Gambar 4. Ilustrasi DFS

(Sumber:

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-\(2025\)-Bahan1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bahan1.pdf)

## 2.4 Bidirectional Search

*Bidirectional Search* adalah strategi pencarian yang bertujuan mempercepat pencarian jalur antara dua simpul dalam graf dengan melakukan dua pencarian secara simultan, satu dari simpul awal menuju simpul tujuan dan satu lagi dari simpul tujuan menuju simpul awal. Konsep ini didasarkan pada prinsip bahwa pencarian dari dua arah akan mengurangi ukuran ruang pencarian secara signifikan dibandingkan dengan pencarian satu arah saja. Dalam banyak kasus, pencarian satu arah memerlukan eksplorasi hingga kedalaman d, sedangkan pencarian dua arah hanya membutuhkan eksplorasi masing-masing hingga kedalaman  $d/2$ , sehingga kompleksitas ruang dan waktu berkang drastis dari  $O(b^d)$  menjadi  $O(b^{(d/2)})$ , dengan b sebagai *branching factor* graf. Biasanya, kedua pencarian menggunakan algoritma *Breadth First Search* (BFS) untuk memastikan bahwa jalur terpendek ditemukan. Tantangan utama dalam *Bidirectional Search* adalah mendeteksi dengan cepat dan efisien saat kedua pencarian bertemu, serta mengelola memori untuk menyimpan semua simpul yang telah dikunjungi dari kedua arah. *Bidirectional Search* banyak digunakan dalam aplikasi seperti pencarian rute terpendek dalam navigasi GPS dan penyelesaian permainan papan yang kompleks.

## 2.5 Aplikasi Web Yang Dibangun

Aplikasi ini merupakan aplikasi berbasis website yang dikembangkan untuk membantu pengguna dalam menemukan kombinasi elemen yang tepat dalam permainan *Little Alchemy 2*. Permainan tersebut mengharuskan pemain menggabungkan elemen-elemen dasar seperti api, air, tanah, dan udara untuk membentuk elemen baru. Seiring bertambahnya elemen, kombinasi yang mungkin semakin kompleks dan sulit ditelusuri secara manual. Oleh karena itu, aplikasi ini memanfaatkan pendekatan algoritmik untuk membantu pencarian tersebut. Sistem pencarian menggunakan dua algoritma pencarian graf populer, yaitu *Breadth First Search* (BFS) dan *Depth First Search* (DFS). Arsitektur aplikasi dibagi menjadi dua bagian utama:

1. **Frontend**, dibangun menggunakan **Next.js** dengan dukungan **TypeScript** untuk menghadirkan antarmuka pengguna yang dinamis, responsif, dan dapat dikembangkan dengan lebih terstruktur. Framework ini memanfaatkan *server side rendering* (SSR) dan optimasi build yang mendukung performa dan SEO.
2. **Backend** dikembangkan dengan menggunakan bahasa **Golang (Go)**. Bahasa ini dipilih karena kemampuannya dalam menangani proses *concurrent* secara efisien, sangat sesuai untuk kebutuhan pencarian resep *multiple* yang melibatkan banyak kombinasi. Backend bertugas mengelola data elemen, menjalankan algoritma pencarian, dan menyediakan API untuk diakses frontend.

Untuk memastikan proses pengembangan dan deployment berjalan lancar serta konsisten di berbagai lingkungan, aplikasi ini juga dikemas menggunakan **Docker**. Dengan Docker, baik frontend maupun backend dapat dijalankan dalam container terisolasi, sehingga memudahkan proses setup dan testing secara lokal maupun saat deployment. Aplikasi ini telah berhasil dideploy secara online. **Frontend** dideploy menggunakan layanan **Vercel**, yang mendukung framework Next.js secara native dan memungkinkan pengelolaan domain serta deployment otomatis dari Git repository. Sementara itu, **backend** dideploy melalui platform **Railway**, yang mendukung container berbasis Docker dan mampu menangani API berbasis Golang dengan efisien.

Data elemen dan kombinasi resep diperoleh melalui proses scraping dari situs Fandom resmi Little Alchemy 2, kemudian diproses dan disimpan dalam struktur data yang memungkinkan pencarian cepat dan validasi tier secara otomatis. Elemen pembentuk resep selalu memiliki tier yang lebih rendah dari elemen hasil. Visualisasi hasil pencarian ditampilkan dalam bentuk pohon (*tree*), yang menunjukkan jalur kombinasi elemen dari elemen dasar hingga ke elemen target. Aplikasi juga menampilkan statistik pencarian seperti durasi proses dan jumlah simpul (*node*) yang dikunjungi. Untuk pencarian *multiple recipe*, sistem menerapkan optimasi *multithreading* agar proses berjalan lebih cepat dan efisien.

## BAB III

### ANALISIS PEMECAHAN MASALAH

#### **3.1 Langkah-Langkah Pemecahan Masalah**

Langkah pertama dalam pemecahan masalah adalah mengumpulkan data elemen dan resep dari permainan Little Alchemy 2. Proses ini dilakukan melalui web scraping terhadap Fandom Wiki Little Alchemy 2 . Kami HTTP client dengan User-Agent yang sesuai untuk menghindari pembatasan akses dari website target. Kemudian, kami mengambil dan melakukan parsing terhadap dokumen HTML dari Fandom Wiki Little wekwek Alchemy 2. Untuk setiap elemen, kami mengekstrak kombinasi elemen yang membentuknya, memastikan setiap resep terdiri dari dua elemen yang valid. Kami juga secara manual menyimpan elemen-elemen dasar sebagai titik awal dalam permainan. Hasil akhir dari proses scraping adalah struktur data yang memetakan setiap elemen ke informasi tier dan resep-resepnya, yang selanjutnya dikonversi ke dalam format RecipeMap dan TierMap untuk penggunaan dalam algoritma pencarian.

Server backend diimplementasikan dengan endpoint “/api/data” yang menerima permintaan berisi target elemen, jenis algoritma, dan parameter pencarian. Berdasarkan input AlgorithmType, server memanggil fungsi MainBfs() atau MainDfs(). Kedua algoritma ini menggunakan pendekatan backward search yang dimulai dari elemen target, menelusuri recipe pembentuknya, dan memastikan hanya elemen dengan tier lebih rendah yang digunakan sebagai komponen.

Untuk optimasi pencarian multiple recipe, kami mengimplementasikan multithreading dengan goroutine. Beberapa goroutine menelusuri jalur penelusuran berbeda secara paralel, dengan hasil dikumpulkan melalui channel. Sistem kami juga menerapkan caching thread-safe untuk menyimpan hasil intermediate, memastikan perhitungan yang sama tidak dilakukan berulang kali oleh thread berbeda. Pendekatan kombinasi multithreading dan caching ini secara signifikan meningkatkan performa untuk pencarian recipe kompleks, terutama pada elemen dengan banyak kombinasi dan jalur pembentukan.

#### **3.2 Pemetaan Masalah Menjadi Elemen-Elemen Algoritma DFS dan BFS**

Pada algoritma DFS dan BFS, dibutuhkan representasi Node pohon tersebut, struktur pohon itu sendiri, dan respons yang diberikan backend ke frontend. Node pohon tersebut dibuat menjadi sebuah struct dengan atribut nama elemen tersebut, kombinasi penggabungan dua elemen, dan children berisi pointer yang mengarah ke ElementNode dari sources tersebut.

Hasil dari respons backend juga dibuat struct dengan atribut target elemen yang dicari, pohon resep, jumlah nodes yang dikunjungi, dan lama waktu pencarian. Jumlah nodes yang dikunjungi dan waktu pencarian merupakan salah satu spesifikasi yang harus ditampilkan.

### 3.2.1 Algoritma BFS

Algoritma BFS pencarian pohon resep menggunakan metode rekursif. Metode rekursif akan memanggil fungsi dirinya sendiri dengan parameter yang sudah diolah. Pada setiap fungsi memasukkan semua gabungan elemen yang menghasilkan elemen tersebut yang didapat dari RecipeMap untuk memastikan pencarian dilakukan secara meluas. Akan tetapi, sebelum dimasukkan akan dipastikan terlebih dahulu tier lebih rendah dari tier parent-nya. Setiap children akan di-*assign* ke worker dari goroutine untuk mempercepat pencarian dengan multithreading.

### 3.2.2 Algoritma DFS

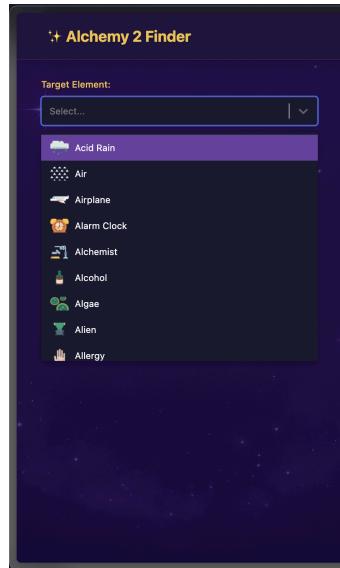
Algoritma DFS dalam pencarian pohon resep juga menggunakan metode rekursif, namun pendekatannya berbeda dengan BFS. Metode rekursif pada DFS akan memanggil fungsi dirinya sendiri untuk mengeksplorasi jalur pencarian secara mendalam sebelum berpindah ke jalur lainnya. Pada setiap pemanggilan fungsi, sistem akan mengambil seluruh kombinasi elemen dari RecipeMap yang dapat membentuk elemen tersebut. Namun, sebelum kombinasi tersebut ditelusuri lebih lanjut, terlebih dahulu akan diperiksa apakah tier dari elemen yang akan ditelusuri lebih rendah dari tier parent-nya, untuk mencegah pencarian yang berputar. DFS akan menelusuri satu cabang pencarian hingga ke elemen paling dasar sebelum kembali (backtrack) dan melanjutkan ke kombinasi lain yang belum dieksplorasi.

## 3.3 Fitur Fungsional dan Arsitektur Aplikasi Web

Aplikasi web ini dirancang untuk membantu pengguna dalam mencari kombinasi pembuatan elemen pada permainan Little Alchemy 2 secara interaktif dan visual. Fitur-fitur utama yang disediakan antara lain:

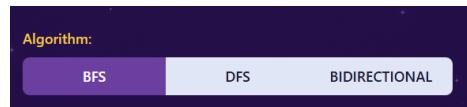
### 1. Pencarian Resep Elemen

Pengguna dapat melakukan pencarian kombinasi elemen untuk membentuk target element yang dipilih. Menu pencarian dilengkapi dengan dropdown interaktif berisi ikon dari setiap elemen, sehingga memudahkan identifikasi dan pemilihan elemen.



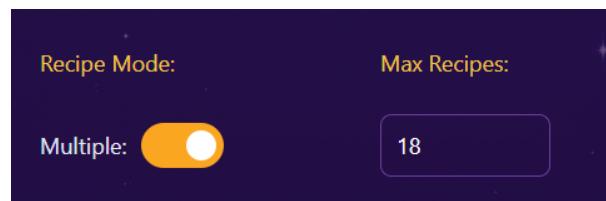
## 2. Pemilihan Algoritma Pencarian

Tersedia dua algoritma pencarian yang dapat dipilih sesuai preferensi pengguna, yaitu BFS (Breadth-First Search), dan DFS (Depth-First Search). Pengguna cukup menekan salah satu tombol algoritma yang tersedia untuk memilih metode pencarian.



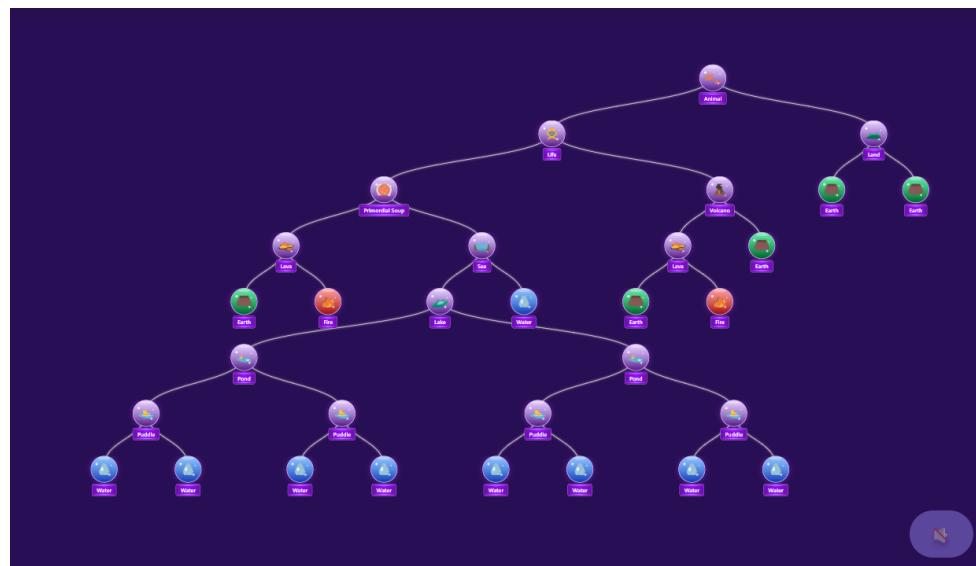
## 3. Mode Multi-Rsep

Aplikasi menyediakan opsi "Multiple" berupa toggle button. Saat diaktifkan, pengguna dapat menentukan jumlah maksimal resep berbeda yang ingin ditampilkan dengan mengisi input Max Recipes. Ini berguna untuk menemukan lebih dari satu jalur kombinasi menuju target elemen.



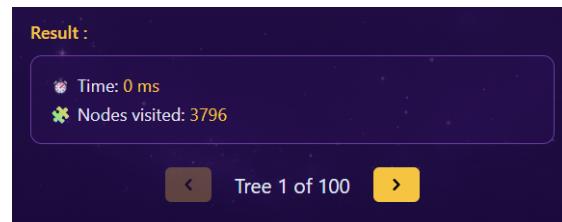
## 4. Visualisasi pohon resep

Hasil pencarian ditampilkan dalam bentuk visualisasi pohon (tree) di sisi kanan layar. Setiap simpul dalam pohon dilengkapi dengan ikon elemen untuk memperjelas langkah-langkah kombinasi. Visualisasi ini membantu pengguna memahami proses pembuatan elemen secara hierarkis.



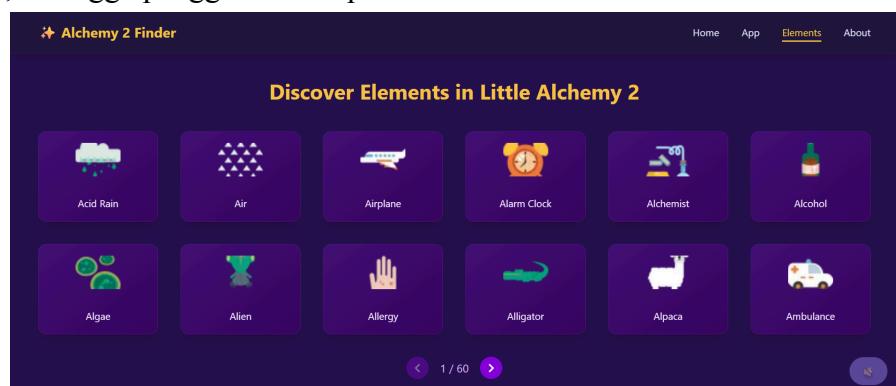
## 5. Statistik Hasil Pencarian

Di bawah tombol “Search”, ditampilkan informasi pencarian berupa waktu eksekusi dan jumlah simpul (node) yang dikunjungi. Jika mode *Multiple* diaktifkan, tersedia pagination yang memungkinkan pengguna untuk melihat seluruh resep yang dihasilkan satu per satu.



## 6. Menu Daftar Elemen

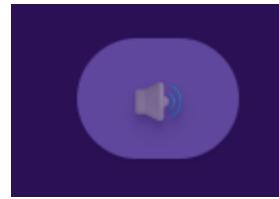
Pada halaman Elements, pengguna dapat melihat daftar lengkap elemen yang tersedia dalam Little Alchemy 2. Daftar ini dilengkapi ikon untuk masing-masing elemen, sehingga pengguna tidak perlu membuka situs resmi atau fandom.



## 7. Fitur Backsound

Untuk menambah pengalaman pengguna, aplikasi menyediakan tombol suara (*sound button*) di pojok kanan bawah. Saat diaktifkan, tombol ini memutar

background bertema magis yang mendukung atmosfer eksploratif saat menggunakan aplikasi.



Arsitektur aplikasi dibagi menjadi dua bagian utama:

1. Frontend

```
.  
  ├── Dockerfile  
  ├── README.md  
  └── doc  
    ├── docker-compose.yml  
    ├── eslint.config.mjs  
    ├── next-env.d.ts  
    ├── next.config.mjs  
    ├── node_modules  
    ├── package-lock.json  
    ├── package.json  
    ├── postcss.config.mjs  
    ├── public  
    └── src  
      ├── app  
      ├── components  
      └── lib  
      └── tsconfig.json
```

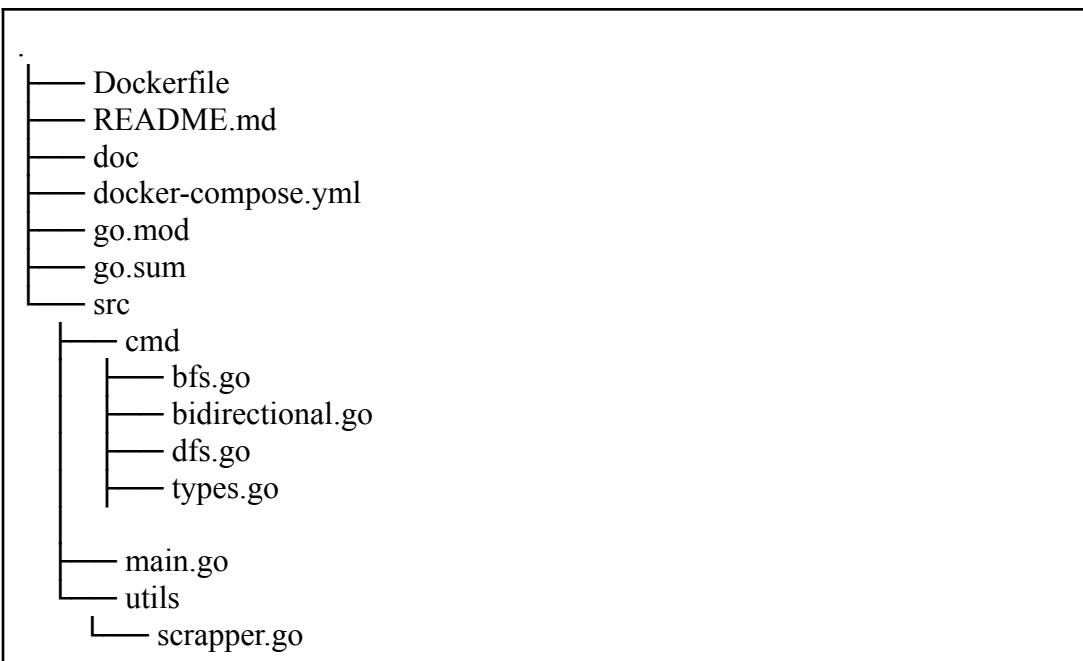
Bagian frontend dikembangkan menggunakan Next.js dengan dukungan TypeScript dan Tailwind CSS. Framework ini dipilih karena mendukung Server-Side Rendering (SSR), optimasi build, dan integrasi yang kuat dengan platform deployment seperti Vercel. Penggunaan TypeScript meningkatkan keandalan dan maintainability kode, sementara Tailwind CSS mempercepat proses styling dengan pendekatan utility-first yang efisien dan konsisten. Salah satu fitur utama frontend adalah visualisasi pohon resep yang dibuat menggunakan library React D3 Tree. Pohon ini menampilkan jalur kombinasi elemen dari elemen dasar hingga target, dilengkapi dengan ikon setiap elemen yang membantu pengguna memahami alur pencampuran.

Secara keseluruhan, frontend memiliki tanggung jawab untuk :

- Menyediakan antarmuka pengguna yang interaktif, responsif, dan mudah digunakan.

- Mengirim permintaan pencarian resep ke backend berdasarkan input pengguna.
- Menampilkan visualisasi pohon resep menggunakan React D3 Tree, termasuk nama dan ikon untuk tiap elemen.
- Menyajikan statistik pencarian, seperti waktu eksekusi dan jumlah simpul (node) yang dikunjungi.
- Mendukung fitur multiple recipe sesuai konfigurasi pengguna (toggle dan jumlah maksimal).
- Dideploy menggunakan Vercel, platform cloud yang mendukung Next.js secara native.

## 2. Backend



Backend dibangun menggunakan bahasa pemrograman Golang (Go) karena kemampuannya dalam menangani proses concurrent secara efisien. Bahasa ini sangat cocok untuk aplikasi yang membutuhkan eksplorasi kombinasi data dalam jumlah besar secara paralel, seperti pada pencarian resep. Untuk mendukung fitur multiple recipe, backend memanfaatkan multithreading dengan goroutine agar proses dapat dilakukan secara paralel dan efisien. Data elemen dan kombinasi resep diperoleh melalui proses scraping dari situs Fandom resmi Little Alchemy 2, lalu disimpan dalam struktur data yang memungkinkan pencarian cepat dan validasi tier secara otomatis.

Secara keseluruhan, backend bertanggung jawab untuk :

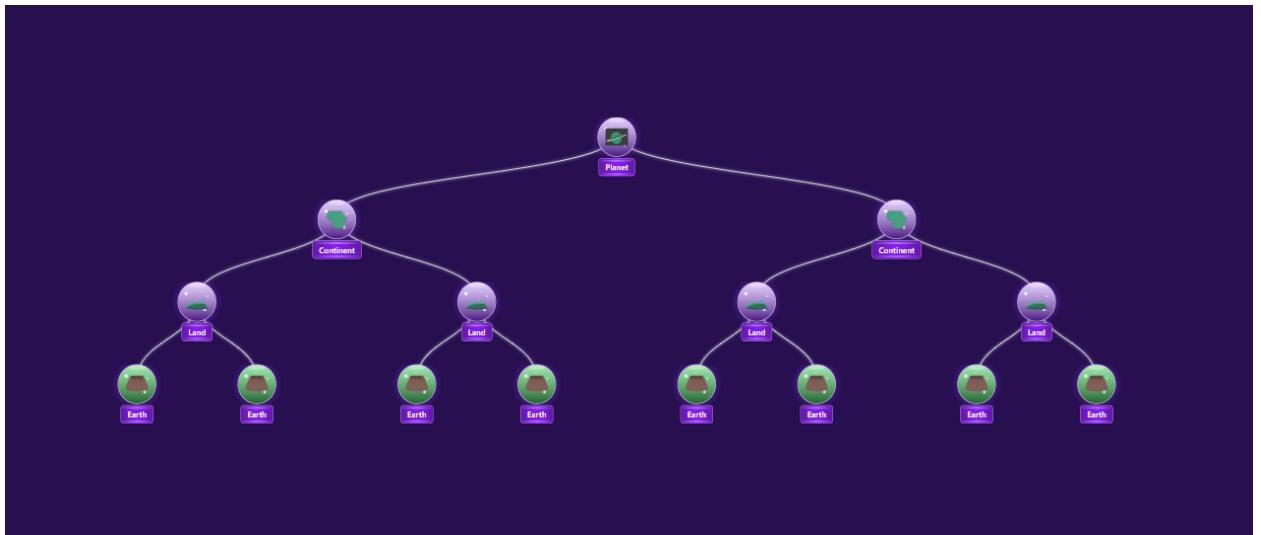
- Mengelola data elemen dan kombinasi resep hasil scraping dari situs Fandom Little Alchemy 2.

- Menjalankan algoritma pencarian (BFS, DFS, dan Bidirectional Search) serta menyusun hasil dalam bentuk struktur pohon.
- Menyediakan API yang dapat diakses frontend untuk menampilkan hasil pencarian.
- Melakukan validasi tier elemen, memastikan bahwa elemen pembentuk memiliki tier lebih rendah dari elemen hasil.
- Mengoptimalkan pencarian multiple recipe menggunakan multithreading (goroutine).
- Dideploy menggunakan Railway, platform cloud yang mendukung container Docker dan cocok untuk menjalankan API berbasis Golang.

### 3.4 Contoh Ilustrasi Kasus

#### 3.4.1 Ilustrasi Kasus BFS

Pada ilustrasi ini , mari kita menggunakan hasil pencarian single recipe planet menggunakan BFS



Pada Algoritma kami , berikut ilustrasinya dalam tabel :

Langkah	Vertex Saat Ini	Queue	Visited
1	Planet	[]	{Planet}
2	Planet	[Continent, Continent]	{Planet}
3	Continent	[Continent]	{Planet, Continent}
4	Continent	[Land, Earth, Continent]	{Planet, Continent}
5	Land	[Earth, Continent]	{Planet, Continent, Land}

6	Land	[Earth, Earth, Earth, Continent]	{Planet, Continent, Land}
7	Earth	[Earth, Earth, Continent]	{Planet, Continent, Land, Earth}
8	Earth	[Earth, Continent]	{Planet, Continent, Land, Earth}
9	Earth	[Continent]	{Planet, Continent, Land, Earth}
10	Continent	[]	{Planet, Continent, Land, Earth}
11	Continent	[Land, Earth]	{Planet, Continent, Land, Earth}
12	Land	[Earth]	{Planet, Continent, Land, Earth}
13	Land	[Earth, Earth, Earth]	{Planet, Continent, Land, Earth}
14	Earth	[Earth, Earth]	{Planet, Continent, Land, Earth}
15	Earth	[Earth]	{Planet, Continent, Land, Earth}
16	Earth	[]	{Planet, Continent, Land, Earth}

Penjelasan Tabel:

1. Mulai dengan vertex Planet sebagai root. Queue kosong dan Planet ditandai sebagai dikunjungi.
2. Mendapatkan kombinasi untuk Planet: [Continent, Continent]. Kedua Continent dimasukkan ke queue.
3. Mengambil Continent pertama dari queue, menandainya sebagai dikunjungi.
4. Mendapatkan kombinasi untuk Continent: [Land, Earth]. Keduanya dimasukkan ke queue.
5. Mengambil Land dari queue, menandainya sebagai dikunjungi.
6. Mendapatkan kombinasi untuk Land: [Earth, Earth]. Keduanya dimasukkan ke queue.

7. Mengambil Earth pertama dari queue, menandainya sebagai dikunjungi. Earth adalah elemen dasar.
8. Mengambil Land berikutnya dari queue, diketahui sudah dikunjungi.
9. Mendapatkan kombinasi untuk Land yang sudah diketahui.
10. Melanjutkan proses hingga semua elemen dalam queue diproses.
11. Queue kosong, pencarian selesai.

Terdapat informasi penting dari hasil bfs algoritma kami , yakni

#### 1. Caching dan Pelacakan Elemen Dikunjungi

Algoritma BFS kami menggunakan set "dikunjungi" untuk melacak elemen yang sudah diproses dan menyimpan recipe yang ditemukan dalam cache. Misalnya , Setelah langkah 7, semua elemen unik (Planet, Continent, Land, Earth) sudah dikunjungi, tetapi proses tetap berlanjut untuk mengeksplorasi semua kombinasi. Mekanisme ini mencegah pengulangan komputasi dan siklus tak terbatas, meningkatkan efisiensi pencarian.

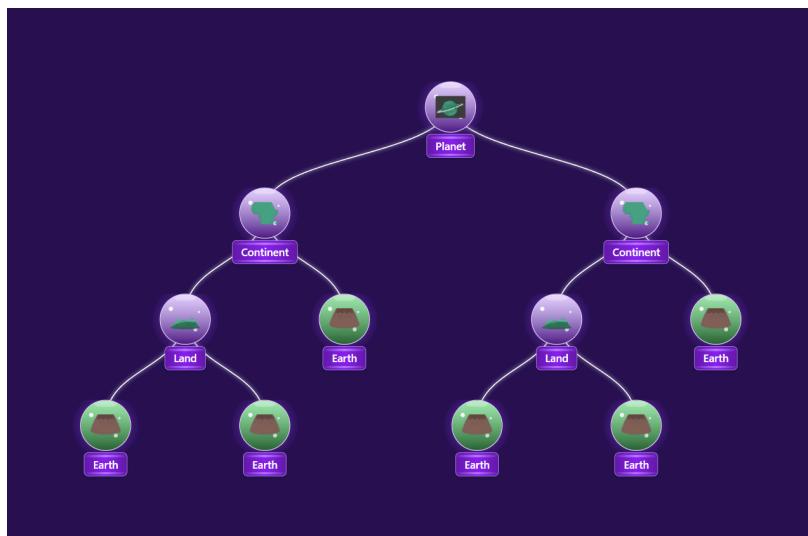
#### 2. Penelusuran Level-by-Level dan Kendali Tier

BFS menelusuri graf level demi level, menjamin recipe terpendek ditemukan terlebih dahulu. Algoritma hanya memproses kombinasi di mana tier elemen komponen lebih rendah dari elemen target. Pendekatan ini sangat efektif untuk menemukan multiple recipe karena mengeksplorasi berbagai jalur secara paralel.

#### 3. Optimasi Performa

Implementasi kami dioptimalkan dengan multithreading untuk mempercepat proses pencarian elemen kompleks.

### 3.4.2 Ilustrasi Kasus BFS



Pada Algoritma kami , berikut tahap pembangunan tree-nya :

- 0: dfsBuildTree(Planet, maxPaths=N, visited={}, depth=0)
  - Planet bukan elemen dasar
  - Belum ada dalam memo cache
  - Mendapatkan kombinasi: [Continent, Continent]
  - Memeriksa tier: parentTier = tier[Planet]
- 1.
  - |— Membuat worker threads dan meneruskan kombinasi ke comboChan
  - |— Worker memproses [Continent, Continent]
  - |— Memeriksa tier: tier[Continent] < tier[Planet] ✓
- 2.
  - |— dfsBuildTree(Continent, visited={Planet}, depth=1)
  - |— Continent bukan elemen dasar
  - |— Belum ada dalam cache
  - |— Mendapatkan kombinasi: [Land, Earth]
  - |— Memeriksa tier: tier[Land] < tier[Continent], tier[Earth] < tier[Continent] ✓
- 3.
  - : |—|— dfsBuildTree(Land, visited={Planet, Continent}, depth=2)
  - |— Land bukan elemen dasar
  - |— Mendapatkan kombinasi: [Earth, Earth]
  - |— Memeriksa tier: tier[Earth] < tier[Land] ✓
- 4:
  - |—|—|— dfsBuildTree(Earth, visited={Planet, Continent, Land}, depth=3)
  - |—|— Earth adalah elemen dasar, mengembalikan node Earth tanpa children
- 5:
  - |—|—|— dfsBuildTree(Earth, visited={Planet, Continent, Land}, depth=3)
  - |—|— Earth adalah elemen dasar, mengembalikan node Earth tanpa children
- 6:
  - |—|— Menggabungkan hasil: Land dengan children [Earth, Earth]
  - |—|— Menyimpan dalam memo cache[Land]
- 7:
  - |—|—|— dfsBuildTree(Earth, visited={Planet, Continent, Land}, depth=2)
  - |—|— Earth adalah elemen dasar, mengembalikan node Earth tanpa children
- 8:
  - |—|— Menggabungkan hasil: Continent dengan children [Land, Earth]
  - |—|— Menyimpan dalam memo cache[Continent]
- 9:
  - |—|—|— dfsBuildTree(Continent, visited={Planet, Continent}, depth=1)
  - |—|— Gunakan hasil dari memo cache[Continent]

- 10:
- |— Menggabungkan hasil: Planet dengan children [Continent, Continent]
  - |— Menyimpan dalam memo cache[Planet]

Algoritma DFS kami melakukan traversal dan backtracking dalam urutan berikut:

1. Planet (elemen target)
2. Continent (komponen pertama)
3. Land (komponen dari Continent)
4. Earth (komponen pertama dari Land) - eksplorasi ke elemen dasar
5. **Backtrack** ke Land setelah Earth selesai dieksplorasi
6. Earth (komponen kedua dari Land) - eksplorasi ke elemen dasar
7. **Backtrack** ke Land setelah semua komponennya selesai dieksplorasi
8. **Backtrack** ke Continent setelah Land selesai
9. Earth (komponen kedua dari Continent) - eksplorasi ke elemen dasar
10. **Backtrack** ke Continent setelah semua komponennya selesai
11. **Backtrack** ke Planet setelah Continent pertama selesai
12. Continent (komponen kedua dari Planet) - menggunakan hasil yang sudah di-cache
13. **Backtrack** ke Planet setelah semua komponennya selesai

Proses backtracking ini terjadi secara implisit dalam implementasi rekursif DFS kita, di mana fungsi `dfsBuildTree()` menyelesaikan satu panggilan rekursif dan kemudian kembali ke panggilan sebelumnya. Pada setiap backtracking, algoritma menyimpan hasil dalam memo cache dan menggunakannya untuk menghindari komputasi berulang di masa mendatang.

Pada pencarian recipe planet , informasi yang dapat diperoleh sebagai berikut :

- Validitas Recipe: Suatu tree recipe dianggap valid jika semua leaf node-nya adalah elemen dasar (air, earth, fire, water). Hal ini terlihat pada langkah 4 dan 5 di mana Earth diidentifikasi sebagai elemen dasar dan menjadi leaf node yang valid.
- Kendali Tier: Proses penelusuran hanya dilanjutkan jika tier elemen komponen lebih rendah dari tier elemen yang dibentuk. Seperti terlihat pada langkah 2 dan 3 di mana algoritma memeriksa `tier[Continent] < tier[Planet]` dan `tier[Land] < tier[Continent]`.
- Kriteria Eksplorasi: Penelusuran rekursif dilanjutkan jika node bukan merupakan elemen dasar dan masih terdapat recipe pembentuknya. Jika node adalah elemen dasar atau tidak memiliki recipe, algoritma akan melakukan backtracking.

- Optimasi dengan Cache: Pencarian tidak diulang untuk elemen yang sama jika sudah pernah ditelusuri sebelumnya, seperti pada langkah 9 di mana hasil untuk Continent langsung diambil dari memo cache.
- Paralelisme: Penggunaan worker threads untuk memproses kombinasi secara paralel meningkatkan efisiensi untuk pencarian multiple recipe, meskipun tidak terlihat langsung dalam trace.

## BAB IV

### IMPLEMENTASI DAN PENGUJIAN

#### 4.1 Spesifikasi Teknis Program

##### 1. Struktur Data

Struktur Data	Penjelasan
<pre>type RecipeMap map[string][][]string</pre>	RecipeMap berupa matrix untuk memetakan resep-resep yang kemungkinan ada
<pre>type TierMap map[string]int</pre>	TierMap memetakan Tier setiap element
<pre>type Result struct {     TargetElement string `json:"target"`     RecipeTree   []ElementNode `json:"tree"`     VisitedNodes int    `json:"nodes"`     SearchTime   float64 `json:"time"` }</pre>	<p>Result berupa json yang dikirimkan sebagai response dari backend ke frontend, dengan atribut TargetElement, VisitedNodes, dan SearchTime.</p> <ul style="list-style-type: none"> <li>- TargetElement untuk memastikan target tersebut sama dengan yang diinginkan,</li> <li>- RecipeTree berupa list resep-resep yang didapatkan,</li> <li>- VisitedNodes adalah banyaknya nodes yang dikunjungi</li> <li>- SearchTime adalah waktu pencarian resep tersebut.</li> </ul>
<pre>type ElementNode struct {     Result string `json:"name"`     Sources []string `json:"sources"`     Children []*ElementNode `json:"children"` }</pre>	<p>ElementNode berfungsi sebagai node dari sebuah tree, dengan atribut</p> <ul style="list-style-type: none"> <li>- Result,</li> <li>- Sources, yaitu banyaknya kombinasi pembuatan</li> <li>- Children, yaitu element yang ada di Sources</li> </ul>
<pre>type MemoCache struct {     mu sync.Mutex     store map[string][]*ElementNode }</pre>	MemoCache berfungsi untuk <i>memoization</i> atau menyimpan resep elemen yang sudah didapatkan supaya mempercepat pencarian yang besar.

##### 2. Fungsi dan Prosedur

###### a. BFS

### MainBfs

Fungsi MainBfs berperan sebagai pemanggilan main.go dalam proses pencarian resep. Fungsi ini menginisialisasi cache untuk menyimpan hasil perhitungan sebelumnya, mencatat waktu proses pencarian, dan memanggil bfsBuildTree untuk membangun pohon resep berdasarkan data resep dan tingkatan (tier) bahan.

```
func MainBfs(recipes RecipeMap, tiers TierMap, target string, maxPaths int) Result {
    var bfsResult Result
    cache := &MemoCache{store: make(map[string][]*ElementNode)}
    startTime := time.Now()
    trees := bfsBuildTree(recipes, tiers, target, maxPaths, cache)
    bfsResult.SearchTime = float64(time.Since(startTime).Milliseconds())
    bfsResult.RecipeTree = flattenTreeList(trees)
    totalNodes := 0
    for _, tree := range bfsResult.RecipeTree {
        totalNodes += countNodes(&tree)
    }
    bfsResult.VisitedNodes = totalNodes

    return bfsResult
}
```

### bfsBuildTree

Fungsi bfsBuildTree merupakan algoritma pencarian pohon resep menggunakan pendekatan rekursif dan paralel. Fungsi ini mengecek apakah target adalah bahan dasar atau sudah ada di cache, lalu mengambil semua kombinasi resep yang memungkinkan. Setiap kombinasi diproses oleh goroutine (worker) yang memanggil bfsBuildTree secara rekursif untuk masing-masing bahan. Hasil kombinasi yang valid digabungkan menjadi node pohon. Proses ini berjalan hingga jumlah jalur maksimum tercapai, kemudian hasil disimpan dalam cache dan dikembalikan ke fungsi utama.

```
func bfsBuildTree(
    recipes RecipeMap,
    tiers TierMap,
    target string,
    maxPaths int,
    cache *MemoCache,
) []*ElementNode {
    var result []*ElementNode

    if isBase(target) {
        node := &ElementNode{
            Result: target,
            Sources: nil,
            Children: nil,
        }
        result = append(result, node)
    }
}
```

```

cache.mu.Lock()
cache.store[target] = []*ElementNode{node}
cache.mu.Unlock()
return []*ElementNode{node}
}

cache.mu.Lock()
if val, ok := cache.store[target]; ok {
    cache.mu.Unlock()
    return val
}
cache.mu.Unlock()

combos, exists := recipes[target]
if !exists {
    return nil
}

parentTier := tiers[target]
queue := make(chan []string, len(combos))
resultsChan := make(chan *ElementNode, maxPaths)
ctx, cancel := context.WithCancel(context.Background())
defer cancel()

var wg sync.WaitGroup

worker := func() {
    defer wg.Done()
    for {
        select {
        case <-ctx.Done():
            return
        case pair, ok := <-queue:
            if !ok {
                return
            }

            if isUnbuildable(pair[0], recipes) || isUnbuildable(pair[1], recipes) {
                continue
            }

            tierA := tiers[pair[0]]
            tierB := tiers[pair[1]]
            if tierA >= parentTier || tierB >= parentTier {
                continue
            }

            // Recursively build children using same memo cache
            leftTrees := bfsBuildTree(recipes, tiers, pair[0], maxPaths, cache)
            rightTrees := bfsBuildTree(recipes, tiers, pair[1], maxPaths, cache)

            for _, left := range leftTrees {
                for _, right := range rightTrees {
                    select {

```

```

        case resultsChan <- &ElementNode{
            Result: target,
            Sources: pair,
            Children: []*ElementNode{left, right},
        }:
        case <-ctx.Done():
            return
        }
    }
}
}

// Launch worker goroutines
workerCount := 2 + parentTier*2
if workerCount > 16 {
    workerCount = 16
}
for i := 0; i < workerCount; i++ {
    wg.Add(1)
    go worker()
}

go func() {
    for _, pair := range combos {
        queue <- pair
    }
    close(queue)
}()

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

for node := range resultsChan {
    result = append(result, node)
    if len(result) >= maxPaths {
        cancel()
        break
    }
}

cache.mu.Lock()
cache.store[target] = result
cache.mu.Unlock()

return result
}

```

b. DFS

## MainDfs

Fungsi MainDfs berperan sebagai pemanggilan dari main.go dalam proses pencarian resep. Fungsi ini bertugas menginisialisasi cache untuk menyimpan hasil perhitungan sebelumnya, mencatat waktu proses pencarian, dan memanggil dfsBuildTree untuk membangun pohon resep berdasarkan data resep dan tingkatan (tier) bahan.

```
func MainDfs(recipes RecipeMap, tiers TierMap, targetElement string, maxRecipes int) Result {  
  
    atomic.StoreInt64(&visitedNodeCount, 0)  
  
    startTime := time.Now()  
    cache := &MemoCache{store: make(map[string][]*ElementNode)}  
  
    if isBaseElement(targetElement) {  
        return Result{  
            TargetElement: targetElement,  
            RecipeTree: []ElementNode{{Result: targetElement}},  
            VisitedNodes: 1,  
            SearchTime: 0,  
        }  
    }  
    trees := dfsBuildTree(recipes, tiers, targetElement, maxRecipes, make(map[string]bool), 0, 15, cache)  
  
    searchTime := float64(time.Since(startTime).Milliseconds())  
  
    result := Result{  
        TargetElement: targetElement,  
        RecipeTree: flattenTreeList(trees),  
        VisitedNodes: int(atomic.LoadInt64(&visitedNodeCount)),  
        SearchTime: searchTime,  
    }  
    return result  
}
```

## dfsBuildTree

Fungsi dfsBuildTree merupakan algoritma pencarian pohon resep menggunakan pendekatan rekursif dan mendalam (depth-first). Fungsi ini mengecek apakah target adalah bahan dasar atau sudah ada di cache, lalu mengambil semua kombinasi resep yang memungkinkan dari RecipeMap. Untuk setiap kombinasi, fungsi memanggil dirinya sendiri (dfsBuildTree) secara rekursif untuk masing-masing bahan dalam kombinasi tersebut, menelusuri satu per satu jalur hingga mencapai kedalaman maksimum atau bahan dasar. Hasil kombinasi yang valid kemudian digabungkan menjadi node pohon resep. Pendekatan ini memungkinkan eksplorasi satu cabang pohon secara menyeluruh sebelum berpindah ke kombinasi lainnya.

Meskipun DFS umumnya berjalan secara sekuensial, proses rekursif ini dapat dioptimalkan dengan menjalankan sebagian pencarian menggunakan goroutine secara terkontrol. Setelah pencarian selesai atau batas jumlah jalur tercapai, hasilnya disimpan ke dalam cache dan dikembalikan ke fungsi utama.

```
func dfsBuildTree(  
    recipes RecipeMap,  
    tiers TierMap,  
    target string,  
    maxPaths int,  
    visited map[string]bool,  
    depth int,  
    maxDepth int,  
    memo *MemoCache,  
) []*ElementNode {  
  
    if isBaseElement(target) {  
        return []*ElementNode{  
            {  
                Result: target,  
                Sources: nil,  
                Children: nil,  
            },  
        }  
    }  
  
    memo.mu.Lock()  
    if val, ok := memo.store[target]; ok {  
        memo.mu.Unlock()  
        return val  
    }  
    memo.mu.Unlock()  
  
    if visited[target] || depth > maxDepth {  
        return []*ElementNode{{Result: target}}  
    }  
  
    combos, exists := recipes[target]  
    if !exists || len(combos) == 0 {  
        return []*ElementNode{{Result: target}}  
    }  
  
    newVisited := make(map[string]bool)  
    for k, v := range visited {  
        newVisited[k] = v  
    }  
    newVisited[target] = true  
  
    parentTier := tiers[target]  
  
    resultsChan := make(chan *ElementNode, maxPaths)  
    ctx, cancel := context.WithCancel(context.Background())
```

```

defer cancel()

comboChan := make(chan []string, len(combos))

var wg sync.WaitGroup
var result []*ElementNode
var resultMutex sync.Mutex

worker := func() {
    defer wg.Done()
    for {
        select {
        case <-ctx.Done():
            return
        case combo, ok := <-comboChan:
            if !ok {
                return
            }

            resultMutex.Lock()
            if len(result) >= maxPaths {
                resultMutex.Unlock()
                return
            }
            resultMutex.Unlock()

            if isUnbuildable(combo[0], recipes) || isUnbuildable(combo[1], recipes) {
                continue
            }

            tierA := tiers[combo[0]]
            tierB := tiers[combo[1]]
            if tierA >= parentTier || tierB >= parentTier {
                continue
            }

            leftTrees := dfsBuildTree(recipes, tiers, combo[0], maxPaths, newVisited, depth+1, maxDepth, memo)
            rightTrees := dfsBuildTree(recipes, tiers, combo[1], maxPaths, newVisited, depth+1, maxDepth, memo)

            for _, left := range leftTrees {
                for _, right := range rightTrees {
                    resultMutex.Lock()
                    if len(result) >= maxPaths {
                        resultMutex.Unlock()
                        return
                    }
                    resultMutex.Unlock()

                    select {
                    case resultsChan <- &ElementNode{
                        Result: target,
                        Sources: combo,
                        Children: []*ElementNode{left, right},
                    }:
                    }
                }
            }
        }
    }
}

```

```

        case <-ctx.Done():
            return
        }
    }
}
}

workerCount := 4
for i:= 0; i < workerCount; i++ {
    wg.Add(1)
    go worker()
}

go func() {
    for _, combo := range combos {
        comboChan <- combo
    }
    close(comboChan)
}()

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

for node := range resultsChan {
    resultMutex.Lock()
    if len(result) >= maxPaths {
        resultMutex.Unlock()
        cancel()
        break
    }
    result = append(result, node)
    resultMutex.Unlock()
}

memo.mu.Lock()
memo.store[target] = result
memo.mu.Unlock()

return result
}

```

c. Utils

isBase
Fungsi isBase memastikan apakah element tersebut apakah base element atau bukan

```
func isBase(e string) bool {  
    return abaseElements[e]  
}
```

### flattenTreeList

Fungsi flattenTreeList mengubah pointer Node menjadi value untuk ditampilkan

```
func flattenTreeList(trees []*ElementNode) []ElementNode {  
    result := make([]ElementNode, len(trees))  
    for i, t := range trees {  
        result[i] = *t  
    }  
    return result  
}
```

### isUnbuildable

Fungsi isUnbuildable memastikan element bisa di build atau tidak dengan elemen yang tidak bisa didapatkan, seperti time dan DLC.

```
func isUnbuildable(e string, recipes RecipeMap) bool {  
    return !isBase(e) && len(recipes[e]) == 0  
}
```

### countNodes

Fungsi countNodes menghitung jumlah node yang dikunjungi dari pohon resep

```
func countNodes(node *ElementNode) int {  
    if node == nil {  
        return 0  
    }  
    count := 1  
    for _, child := range node.Children {  
        count += countNodes(child)  
    }  
    return count  
}
```

## 4.2 Tata Cara Penggunaan Program

Berikut akan dijelaskan langkah-langkah yang perlu dilakukan untuk menjalankan program, baik secara lokal maupun menggunakan Docker. Program ini dibagi menjadi

dua bagian utama, yaitu frontend (antarmuka pengguna berbasis Next.js) dan backend (server aplikasi berbasis bahasa Go). Berikut langkah-langkah yang perlu dilakukan untuk menjalankan program baik secara lokal menggunakan Docker atau tanpa menggunakan Docker, dan secara online melalui alamat hasil *deployment*.

- Menjalankan Program Tanpa Docker

Untuk menjalankan aplikasi tanpa Docker, pengguna harus menginstal beberapa perangkat lunak terlebih dahulu, yaitu Node.js dan npm untuk menjalankan bagian frontend, dan Go (versi minimal 1.18) untuk menjalankan backend.

1. Menjalankan Frontend

- 1) Kloning repositori frontend dari GitHub:

```
git clone https://github.com/carllix/Tubes2_FE_BFC
```

- 2) Masuk ke direktori hasil kloning:

```
cd Tubes2_FE_BFC
```

- 3) Instal seluruh dependensi yang dibutuhkan:

```
npm install
```

- 4) Buat file konfigurasi environment lokal untuk mengatur alamat backend:

```
NEXT_PUBLIC_BACKEND_URL=http://localhost:8080
```

- 5) Jalankan server pengembangan Next.js:

```
npm run dev
```

2. Menjalankan Backend

- 1) Kloning repositori backend dari GitHub:

```
git clone https://github.com/barruadi/Tubes2_BE_BFC
```

- 2) Masuk ke direktori backend:

```
cd Tubes2_BE_BFC
```

- 3) Unduh seluruh dependensi dengan perintah:

```
go mod download
```

- 4) Jalankan server backend:

```
go run ./src
```

Setelah kedua server (frontend dan backend) berhasil dijalankan, pengguna dapat membuka aplikasi pada peramban dengan mengakses alamat <http://localhost:3000>.

- Menjalankan Program dengan Docker

Cara alternatif yang lebih praktis adalah dengan menggunakan Docker untuk menghindari konflik dependensi dan menjamin konsistensi lingkungan pengembangan. Pastikan Docker Desktop sudah terinstal dan aktif sebelum menjalankan langkah-langkah berikut.

1. Menjalankan Frontend

- 1) Kloning repositori frontend:

```
git clone https://github.com/carllix/Tubes2_FE_BFC
```

- 2) Masuk ke direktori frontend:

```
cd Tubes2_FE_BFC
```

- 3) Buat file konfigurasi production:

```
NEXT_PUBLIC_BACKEND_URL=http://localhost:8080
```

- 4) Build dan jalankan container frontend:

```
docker compose up --build
```

2. Menjalankan Backend

- 1) Kloning repositori backend:

```
git clone https://github.com/barruadi/Tubes2_BE_BFC
```

- 2) Masuk ke direktori backend:

```
cd Tubes2_BE_BFC
```

- 3) Build dan jalankan container backend:

```
docker compose up --build
```

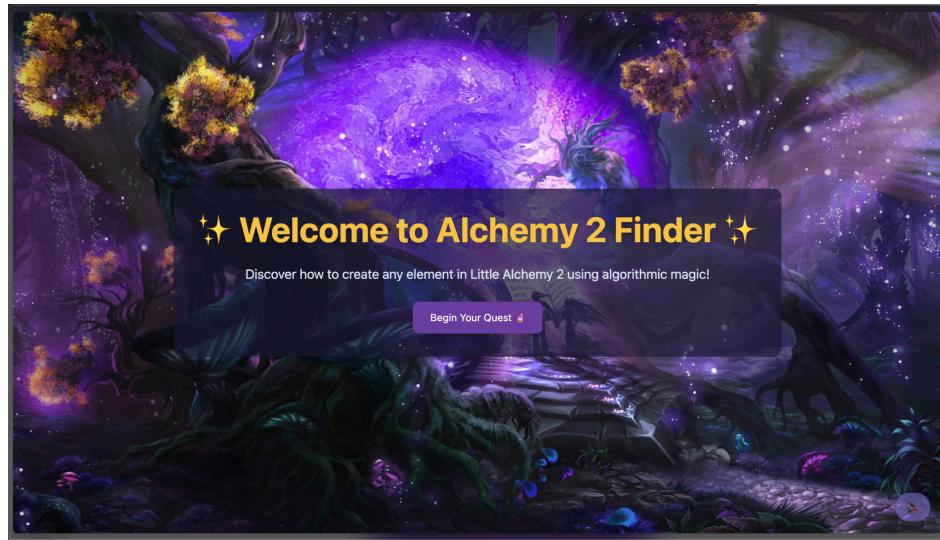
Setelah kedua container aktif, pengguna dapat membuka aplikasi pada peramban dengan mengakses alamat <http://localhost:3000>.

- Menjalankan Program yang Telah Dideploy

Untuk mempermudah pengguna tanpa perlu mengatur *local environment*, aplikasi ini telah dideploy secara online. Versi ini dapat langsung diakses melalui alamat:

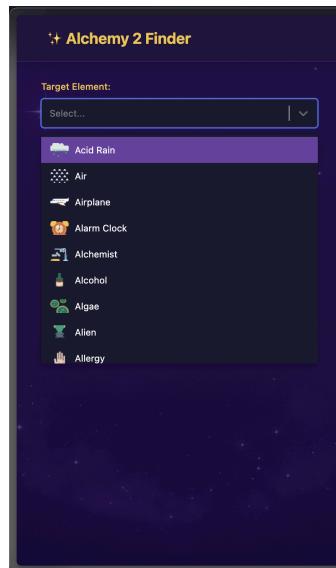
- Frontend: <https://tubes2-fe-bfc.vercel.app>
- Backend: <https://tubes2bebfc-production.up.railway.app/>

Setelah berhasil menjalankan program, pengguna akan melihat tampilan Home Page/Landing Page sebagai berikut:

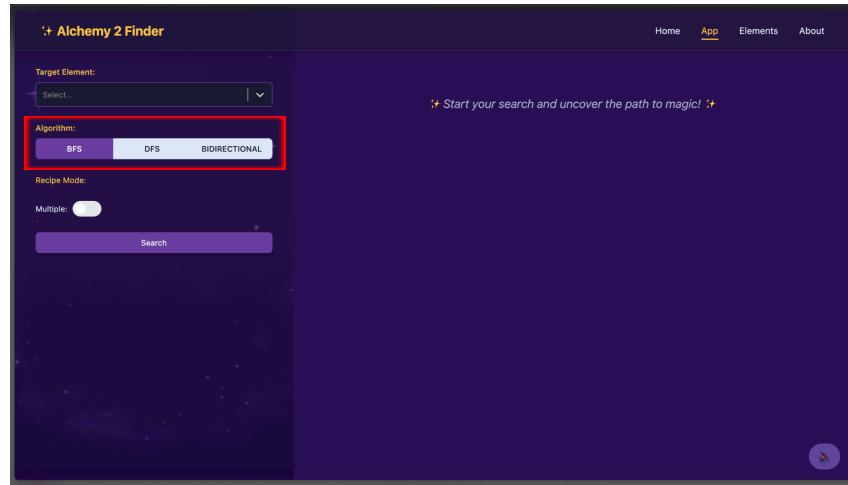


Berikut adalah tata cara penggunaan program Alchemy 2 Recipe Finder:

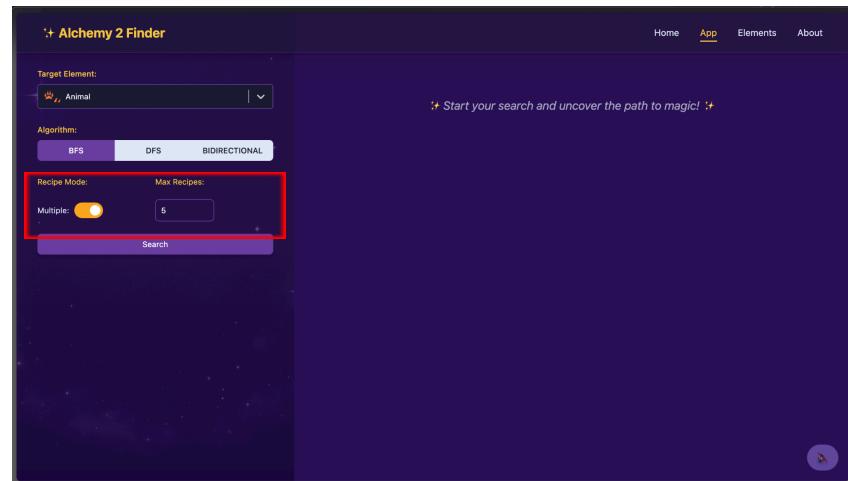
1. Klik tombol “**Begin Your Quest**” untuk menuju ke halaman **App**, yaitu halaman tempat pengguna dapat melakukan pencarian resep elemen.
2. Setelah berada di halaman App, pengguna dapat memilih **elemen tujuan (Target Element)** yang ingin dicari resepnya.



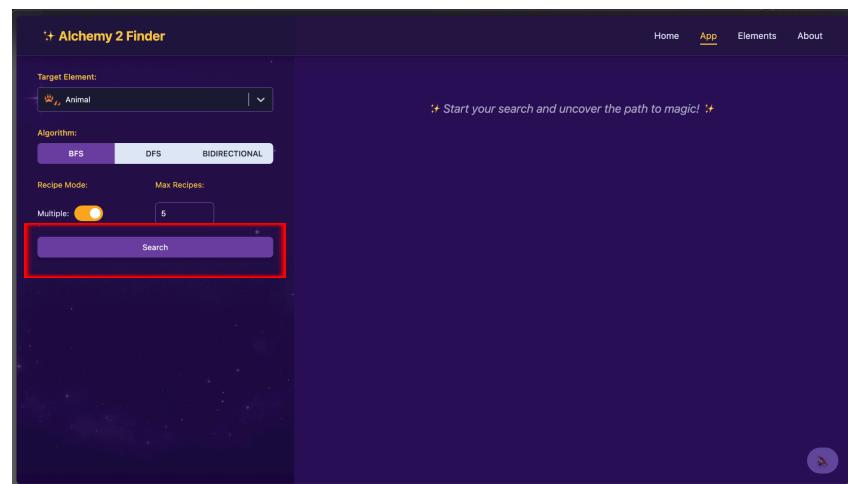
3. Selanjutnya, pilih salah satu **algoritma pencarian** yang tersedia, yaitu **BFS** atau **DFS**



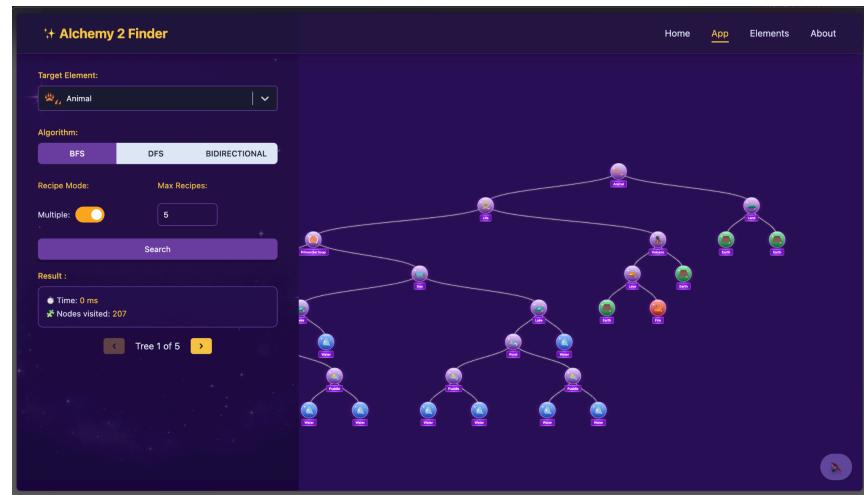
4. Pengguna dapat mengaktifkan **toggle button** "Multiple" jika ingin mendapatkan lebih dari satu resep. Jika opsi ini diaktifkan, pengguna harus mengisi jumlah maksimal resep yang ingin ditampilkan.



5. Setelah semua opsi dipilih, tekan tombol “Search” untuk memulai proses pencarian.



6. Hasil pencarian akan ditampilkan dalam bentuk **tree visualisasi** pada kolom sebelah kanan, disertai informasi berupa **waktu eksekusi** dan **Jumlah node yang dikunjungi**.



### 4.3 Hasil Pengujian

Pengujian dilakukan untuk menentukan algoritma pencarian yang paling efektif dalam mencari resep dari suatu target element, berdasarkan jumlah node yang dikunjungi dan waktu eksekusi. Pengujian ini dilakukan terhadap 3 elemen target berbeda dengan menggunakan 3 algoritma yang berbeda yaitu BFS dan DFS.

Tabel 1 Hasil Pengujian Element Airplane

Algoritma	Mode	
	Single	Multiple (Max Recipe: 10)

## BFS

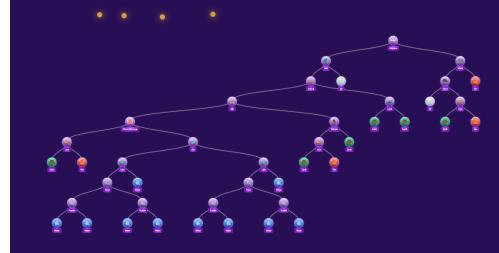
Target Element: Airplane

Algorithm: **BFS** DFS BIDIRECTIONAL

Recipe Mode:  Multiple:

Search

Result :  
⌚ Time: 22738 µs  
✿ Nodes visited: 701



Target Element: Airplane

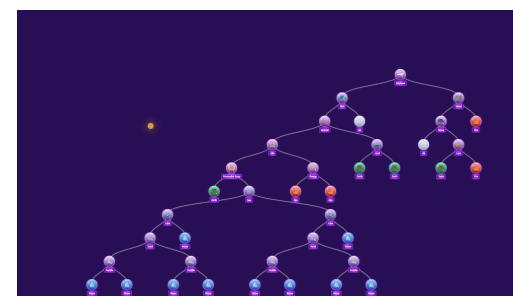
Algorithm: **BFS** DFS BIDIRECTIONAL

Recipe Mode:  Max Recipes:  10

Search

Result :  
⌚ Time: 1737 µs  
✿ Nodes visited: 410

N Tree 1 of 10 >



## DFS

Alchemy 2 Finder

Target Element: Airplane

Algorithm: **DFS** BFS BIDIRECTIONAL

Recipe Mode:  Multiple:

Search

Result :  
⌚ Time: 18632 µs  
✿ Nodes visited: 450

Alchemy 2 Finder

Target Element: Airplane

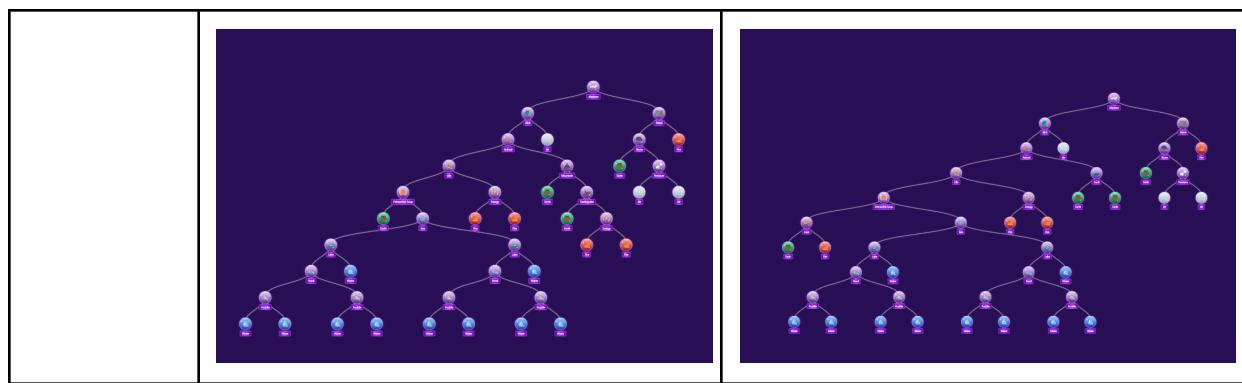
Algorithm: **DFS** BFS BIDIRECTIONAL

Recipe Mode:  Max Recipes:  10

Search

Result :  
⌚ Time: 6835 µs  
✿ Nodes visited: 430

N Tree 1 of 10 >



Tabel 2 Hasil Pengujian Element Picnic

Algoritma	Mode	
	Single	Multiple (Max Recipe: 100)
BFS	<p>Target Element:</p> <input type="text" value="Picnic"/> <p>Algorithm:</p> <input checked="" type="radio"/> BFS <input type="radio"/> DFS <input type="radio"/> BIDIRECTIONAL	

Recipe Mode:

Multiple:

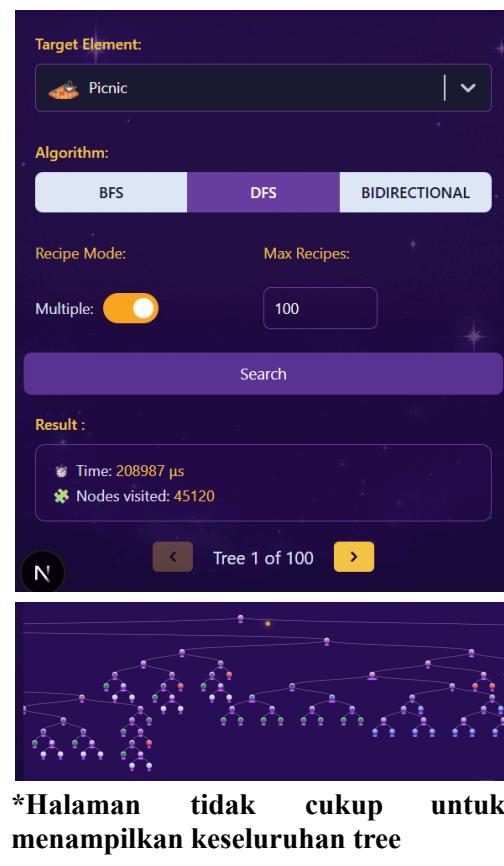
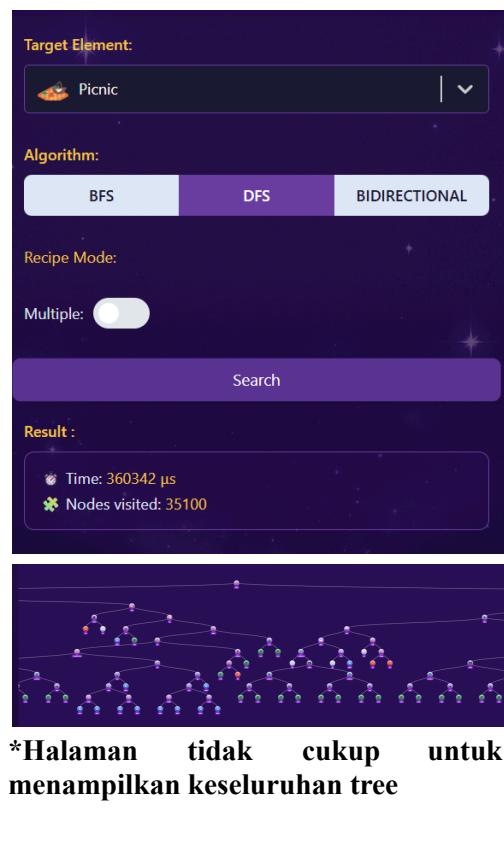
**Search**

Result :

- ⌚ Time: 122254 µs
- ✳️ Nodes visited: 2384

\*Halaman tidak cukup untuk menampilkan keseluruhan tree

DFS



Tabel 3 Hasil Pengujian Element Animal

Algoritma	Mode	
	Single	Multiple (Max Recipe: 1000)

## BFS

Target Element:

Algorithm:  BFS  DFS  BIDIRECTIONAL

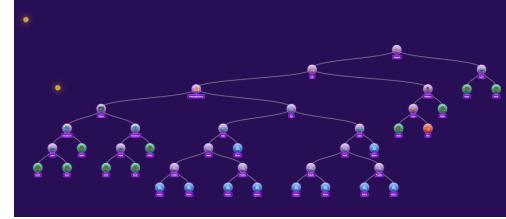
Recipe Mode:

Multiple:

Search

Result :

⌚ Time: 3973  $\mu$ s  
✿ Nodes visited: 4532



Target Element:

Algorithm:  BFS  DFS  BIDIRECTIONAL

Recipe Mode: Max Recipes:

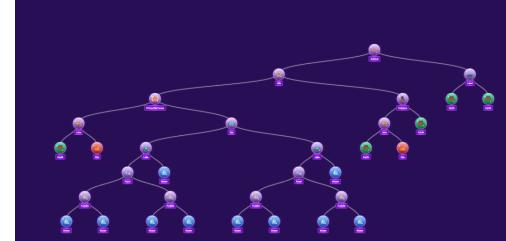
Multiple:  1000

Search

Result :

⌚ Time: 17371  $\mu$ s  
✿ Nodes visited: 47100

N Tree 1 of 1000 >



## DFS

Target Element:

Algorithm:  BFS  DFS  BIDIRECTIONAL

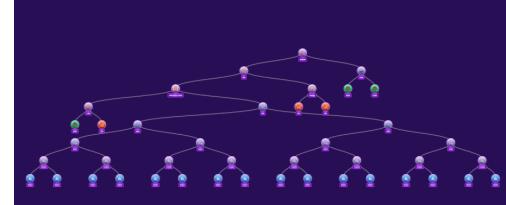
Recipe Mode:

Multiple:

Search

Result :

⌚ Time: 17190  $\mu$ s  
✿ Nodes visited: 48012



Target Element:

Algorithm:  BFS  DFS  BIDIRECTIONAL

Recipe Mode: Max Recipes:

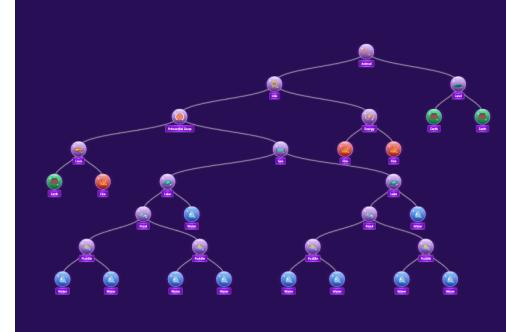
Multiple:  1000

Search

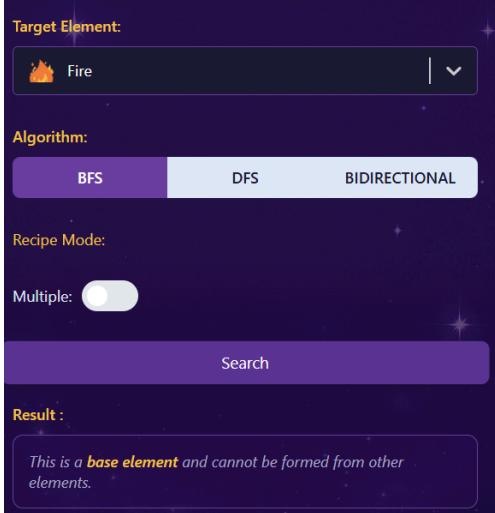
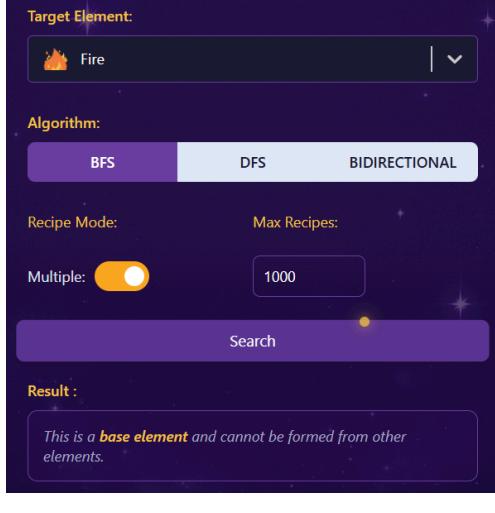
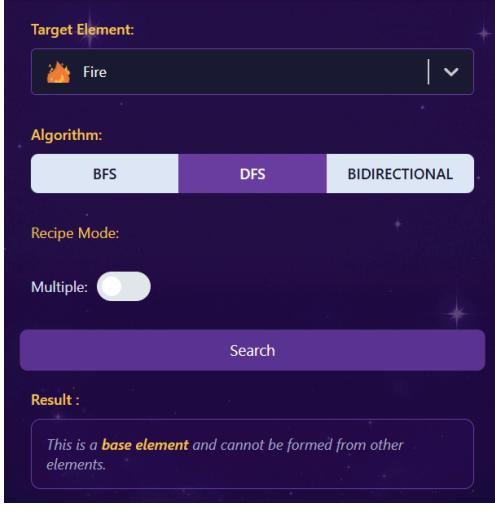
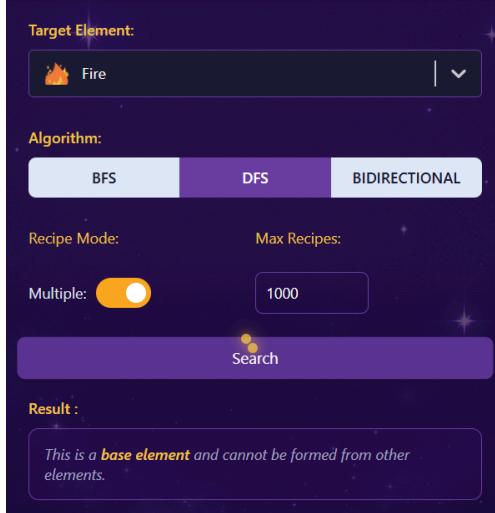
Result :

⌚ Time: 5872  $\mu$ s  
✿ Nodes visited: 43536

N Tree 1 of 1000 >



Tabel 4 Hasil Pengujian Element Fire (Best Element)

Algoritma	Mode	
	Single	Multiple (Max Recipe: 1000)
BFS	 <p>The screenshot shows the search interface for BFS mode. The target element is set to 'Fire'. The algorithm selected is 'BFS'. The search results indicate that 'Fire' is a base element and cannot be formed from other elements.</p>	 <p>The screenshot shows the search interface for BFS mode with the 'Multiple' option enabled. The target element is 'Fire', and the maximum number of recipes is set to 1000. The search results are identical to the single search, stating that 'Fire' is a base element.</p>
DFS	 <p>The screenshot shows the search interface for DFS mode. The target element is set to 'Fire'. The algorithm selected is 'DFS'. The search results indicate that 'Fire' is a base element and cannot be formed from other elements.</p>	 <p>The screenshot shows the search interface for DFS mode with the 'Multiple' option enabled. The target element is 'Fire', and the maximum number of recipes is set to 1000. The search results are identical to the single search, stating that 'Fire' is a base element.</p>

Tabel 5 Hasil Pengujian Element Alarm Clock (Element yang menggunakan Time)

Algoritma	Mode	
	Single	Multiple (Max Recipe: 1000)

BFS

Target Element:

 Alarm Clock | ▾

Target Element:

 Alarm Clock | ▾

DFS

Target Element:

 Alarm Clock | ▾

Target Element:

 Alarm Clock | ▾

#### 4.4 Analisis Hasil Pengujian

Berdasarkan hasil pengujian yang telah dilakukan, dapat disimpulkan bahwa dari segi waktu eksekusi, perbedaan antara algoritma BFS dan DFS relatif tidak terlalu signifikan. Pada kasus elemen *Airplane*, algoritma DFS menunjukkan performa yang sedikit lebih cepat dibandingkan BFS. Sebaliknya, pada elemen lain, BFS cenderung memiliki waktu eksekusi yang lebih cepat dibandingkan DFS. Hal ini menunjukkan bahwa tidak ada algoritma pencarian yang secara mutlak lebih unggul dalam semua kasus. Kecepatan masing-masing algoritma sangat bergantung pada karakteristik dari elemen target serta struktur jalur pencarinya. Selain itu, efisiensi waktu juga dipengaruhi oleh penggunaan *goroutine* untuk multithreading yang memungkinkan eksekusi secara paralel, serta pemanfaatan *memory caching* yang mempercepat proses pencarian.

Dari sisi jumlah simpul yang dikunjungi, terlihat bahwa pada kasus elemen *Airplane*, DFS mengunjungi simpul lebih sedikit dibandingkan BFS, sedangkan pada elemen lainnya BFS lebih efisien dalam jumlah simpul yang dikunjungi. Hal ini menunjukkan bahwa dalam kasus tertentu seperti *Airplane*, algoritma DFS lebih cocok digunakan karena lebih hemat dalam eksplorasi simpul. Namun, pada kebanyakan kasus lainnya, BFS mampu memberikan hasil dengan kunjungan simpul yang lebih sedikit sehingga dapat dikatakan lebih efisien dalam konteks tersebut. Efektivitas algoritma sangat dipengaruhi oleh struktur dan arah kombinasi elemen yang tersedia dalam *recipe map*.

Dari segi visualisasi pohon yang dihasilkan, algoritma BFS dan DFS menunjukkan perbedaan yang cukup terlihat. Algoritma BFS menghasilkan pohon yang cenderung melebar karena pendekatan pencarinya yang mengeksplorasi semua simpul pada satu tingkat sebelum melanjutkan ke tingkat berikutnya. Sebaliknya, algoritma DFS menghasilkan struktur pohon yang lebih dalam karena menelusuri satu jalur secara mendalam hingga tidak memungkinkan lagi, baru kemudian melakukan *backtrack*. Hal ini sesuai dengan prinsip dasar dari masing-masing algoritma: BFS menggunakan struktur data *queue*, sementara DFS menggunakan *stack*.

Untuk pencarian dengan opsi *multiple recipe*, baik BFS maupun DFS menghasilkan jumlah pohon resep yang sama. Hal ini menunjukkan bahwa perbedaan algoritma pencarian tidak mempengaruhi banyaknya kombinasi resep yang ditemukan, melainkan hanya memengaruhi jalur pencarian, jumlah simpul yang dikunjungi, dan struktur visualisasi yang terbentuk. Jumlah pohon yang dihasilkan dalam mode multiple lebih ditentukan oleh parameter maksimum resep yang diberikan dan ketersediaan jalur kombinasi dalam data *recipe map*, bukan oleh jenis algoritmanya.

Pada kasus pengujian keempat, yaitu elemen *Fire*, algoritma tidak dapat membentuk pohon resep karena *Fire* merupakan salah satu *base element* dalam permainan Little Alchemy 2. Oleh karena itu, elemen ini tidak memiliki kombinasi pembentuk dan tidak memerlukan pencarian. Sementara itu, pada kasus kelima dengan elemen *Alarm Clock*, pohon resep juga tidak ditemukan karena elemen ini membutuhkan elemen *Time* sebagai

salah satu komponennya. Dalam konteks tugas ini, elemen *Time* dianggap sebagai elemen spesial yang tidak dapat dihasilkan dari kombinasi elemen apa pun. Dengan demikian, semua elemen yang membutuhkan *Time* sebagai bagian dari resepnya tidak akan dapat ditemukan pohon resepnya.

## BAB V

### KESIMPULAN, SARAN, DAN REFLEKSI

#### 5.1 Kesimpulan

Melalui Tugas Besar 2 IF2211 Strategi Algoritma, kami telah mengembangkan aplikasi berbasis website yang dirancang untuk membantu pengguna dalam menemukan kombinasi elemen yang tepat pada permainan *Little Alchemy 2*. Aplikasi ini dibangun menggunakan Next.js dengan TypeScript untuk bagian frontend, dan Golang untuk bagian backend. Seluruh komponen aplikasi dikemas menggunakan Docker untuk mempermudah pengembangan lintas lingkungan dan proses deployment. Aplikasi ini telah berhasil dideploy secara online, di mana frontend menggunakan Vercel dan backend menggunakan Railway.

Dalam proses pencarian resep elemen, aplikasi ini mengimplementasikan dua algoritma pencarian, yaitu Breadth First Search (BFS) dan Depth First Search (DFS). Berdasarkan hasil pengujian, performa kedua algoritma cukup seimbang, dengan keunggulan masing-masing tergantung pada elemen yang dicari. DFS lebih efisien pada kasus tertentu seperti *Airplane*, sedangkan BFS lebih optimal untuk sebagian besar elemen lainnya. Dari sisi jumlah simpul yang dikunjungi dan visualisasi pohon, BFS cenderung menghasilkan pohon melebar dan eksplorasi lebih luas, sementara DFS membentuk pohon lebih dalam dengan eksplorasi mendalam. Pada mode *multiple recipe*, keduanya menghasilkan jumlah pohon yang sama. Beberapa elemen seperti *Fire* dan *Alarm Clock* tidak dapat ditemukan karena bersifat dasar atau melibatkan elemen spesial seperti *Time*. Hal ini menunjukkan bahwa efektivitas algoritma sangat dipengaruhi oleh karakteristik elemen dan struktur data yang digunakan.

#### 5.2 Saran

Dalam pelaksanaan tugas besar ini, terdapat beberapa saran yang dapat dipertimbangkan untuk pengembangan program di masa depan:

1. Tidak menunda-nunda dalam mengerjakan Tugas Besar agar dapat lebih maksimal dalam menyelesaikan dan mengeksplorasi seluruh spesifikasi wajib dan bonus.
2. Menetapkan kontrak data antara backend dan frontend sejak awal, seperti skema respons API, untuk menghindari *mismatch* data saat proses integrasi.
3. React D3 Tree perlu dieksplorasi lebih mendalam untuk mengoptimalkan tampilan dan estetika struktur tree yang divisualisasikan di aplikasi.

#### 5.3 Refleksi

Dalam pelaksanaan Tugas Besar 2 ini, kami belajar bahwa implementasi algoritma pencarian seperti BFS dan DFS, bukan hanya soal teori semata, tetapi juga sangat berkaitan dengan bagaimana data direpresentasikan dan dikelola dalam sistem nyata. Tantangan utama yang kami hadapi bukan hanya dalam menulis kode algoritmanya,

melainkan juga dalam merancang struktur data yang efisien, memastikan validasi tier elemen, dan mengatur komunikasi antara frontend dan backend agar berjalan lancar. Penerapan teknologi seperti Docker juga membuka wawasan kami mengenai pentingnya *environment consistency* dalam pengembangan aplikasi modern, terutama saat bekerja dalam tim. Selain itu, proses *scraping* data dari situs Fandom dan mengonversinya menjadi format yang bisa diproses oleh backend memberi pengalaman dalam praktik dasar *data engineering*.

Bekerja dalam kelompok mengajarkan kami pentingnya kolaborasi, komunikasi yang jelas, serta pembagian tugas yang efektif. Kami juga menyadari bahwa eksplorasi lebih lanjut terhadap *tools* seperti *React D3 Tree* dapat memberikan nilai tambah tidak hanya secara fungsional tetapi juga dalam hal *user experience*. Secara keseluruhan, tugas besar ini tidak hanya memperkuat pemahaman kami terhadap strategi algoritma pencarian, tetapi juga memperluas kemampuan teknis kami dalam membangun dan mengintegrasikan sistem aplikasi berbasis web secara menyeluruh.

## LAMPIRAN

### Tautan Repository Github

Berikut tautan repository github kelompok kami untuk Tugas Besar 2 IF2211 Strategi Algoritma :

- **Frontend** : [https://github.com/carllix/Tubes2\\_FE\\_BFC](https://github.com/carllix/Tubes2_FE_BFC)
- **Backend** : [https://github.com/barruadi/Tubes2\\_BE\\_BFC](https://github.com/barruadi/Tubes2_BE_BFC)

### Tautan Video

Berikut tautan video kelompok kami untuk Tugas Besar 2 IF2211 Strategi Algoritma :  
<https://youtu.be/6pqk5PlPjLk?si=Sw2G9T-tUfoCv972>

### Tabel Kelengkapan Spesifikasi

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

- Amazon Web Services. 2025. *What is an API?* <https://aws.amazon.com/what-is/api/> (Diakses pada 07 Mei 2025).
- Gin Contributors. 2025. *Gin*. <https://gin-gonic.com/> (Diakses pada 07 Mei 2025).
- MDN Web Docs. 2025. *Client-Server Architecture*. [https://developer.mozilla.org/en-US/docs/Learn\\_web\\_development/Extensions/Server-side/First\\_steps/Client-Server\\_overview](https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/Client-Server_overview) (Diakses pada 07 Mei 2025).
- Meta. 2025. *React Documentation*. <https://react.dev/> (Diakses pada 07 Mei 2025).
- Munir, Rinaldi. 2025. *Breadth/Depth First Search (BFS/DFS) (Bagian 1)*. [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf) (Diakses pada 07 Mei 2025).
- Munir, Rinaldi. 2025. *Breadth/Depth First Search (BFS/DFS) (Bagian 2)*. [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-\(2025\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-(2025)-Bagian2.pdf) (Diakses pada 07 Mei 2025).
- PuerkitoBio. 2025. *goquery*. <https://github.com/PuerkitoBio/goquery> (Diakses pada 07 Mei 2025).
- The Go Authors. 2025. *Effective Go: Concurrency*. [https://go.dev/doc/effective\\_go#concurrency](https://go.dev/doc/effective_go#concurrency) (Diakses pada 07 Mei 2025).
- The Go Authors. 2025. *Golang Documentation*. <https://go.dev/doc/> (Diakses pada 07 Mei 2025).
- Vercel. 2025. *Next Documentation*. <https://nextjs.org/docs> (Diakses pada 07 Mei 2025).