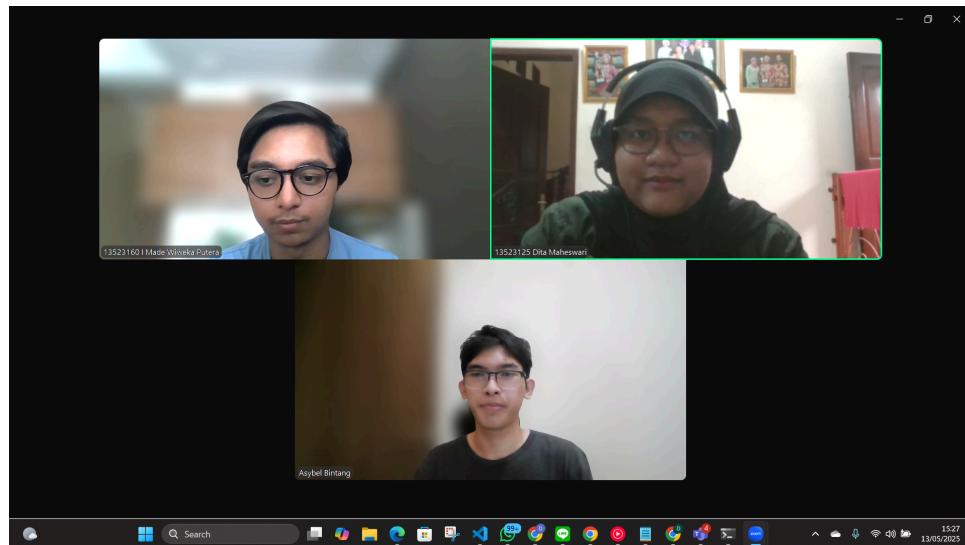


**LAPORAN TUGAS BESAR 2 IF2211 STRATEGI ALGORITMA**  
**SEMESTER II TAHUN 2024/2025**

**PEMANFAATAN ALGORITMA BFS DAN DFS DALAM PENCARIAN RECIPE  
PADA PERMAINAN LITTLE ALCHEMY 2**

Kelompok: SemogaGaMasukUGD



Disusun Oleh:

Dita Maheswari (13523125)

I Made Wiweka Putera (13523160)

Asybel B.P. Sianipar (15223011)

**PROGRAM STUDI TEKNIK INFORMATIKA**  
**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**  
**INSTITUT TEKNOLOGI BANDUNG**

**2025**

## DAFTAR ISI

<b>DAFTAR ISI.....</b>	<b>1</b>
<b>BAB I – DESKRIPSI MASALAH.....</b>	<b>2</b>
<b>BAB II – LANDASAN TEORI.....</b>	<b>4</b>
2.1. Penjelajahan Graf.....	4
2.2. Breadth First Search (BFS).....	4
2.3. Depth First Search (DFS).....	6
2.4. Aplikasi Web.....	8
<b>BAB III – ANALISIS PEMECAHAN MASALAH.....</b>	<b>10</b>
3.1. Langkah Pemecahan Masalah.....	10
3.1.1. Pendefinisian Masalah.....	10
3.1.3. Implementasi Solusi.....	11
3.1.3 Integrasi Backend dan Frontend.....	11
3.2. Pemetaan Masalah.....	12
3.2.1. Pemetaan Masalah menjadi Elemen Algoritma BFS.....	13
3.2.2. Pemetaan Masalah menjadi Elemen Algoritma DFS.....	16
3.3. Arsitektur Aplikasi Web.....	18
3.4. Ilustrasi Kasus.....	20
3.4.1. BFS.....	20
3.4.2. DFS.....	23
<b>BAB IV – IMPLEMENTASI DAN PENGUJIAN.....</b>	<b>24</b>
4.1. Spesifikasi Teknis dan Struktur Data Program.....	24
4.2. Implementasi program.....	31
4.3. Tata Cara Penggunaan Program.....	51
4.4. Hasil Pengujian.....	55
4.5. Analisis Pengujian.....	59
<b>BAB V – KESIMPULAN, SARAN, DAN REFLEKSI TUGAS BESAR 2.....</b>	<b>61</b>
5.1. Kesimpulan.....	61
5.2. Saran.....	61
5.3. Refleksi Tugas Besar 2.....	61
<b>DAFTAR PUSTAKA.....</b>	<b>62</b>
<b>LAMPIRAN.....</b>	<b>63</b>
A. Pranala Repository:.....	63
B. Pranala Video:.....	63
C. Tabel Ketercapaian.....	63

## BAB I – DESKRIPSI MASALAH

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

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

Pada Tugas Besar kedua 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. Penjelajahan Graf

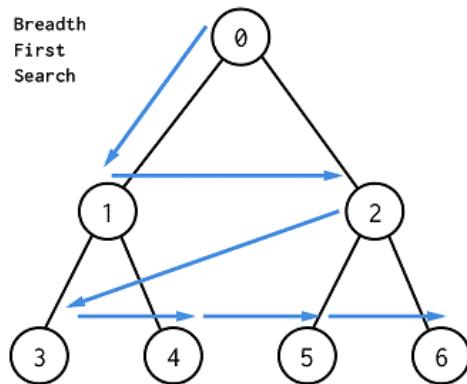
Graf didefinisikan sebagai struktur data yang terdiri dari himpunan simpul (vertices) dan himpunan sisi (edges) yang menghubungkan simpul-simpul tersebut. Secara formal, graf G dinyatakan sebagai  $G = (V, E)$ , di mana  $V$  adalah himpunan simpul dan  $E$  adalah himpunan sisi.

Penelusuran graf adalah proses mengunjungi atau memeriksa setiap simpul (vertex) dan sisi (edge) dalam struktur data graf secara sistematis. Tujuannya adalah untuk memproses data yang tersimpan di simpul-simpul graf atau mencari simpul tertentu). Penelusuran graf digunakan dalam berbagai aplikasi, seperti mencari rute terpendek, analisis jaringan, atau pemrosesan data relasional.

Dalam konteks pencarian solusi, algoritma penelusuran graf dibagi menjadi pencarian tanpa informasi (uninformed search) dan pencarian dengan informasi (informed search). Pencarian tanpa informasi, seperti BFS, DFS, Depth Limited Search, Iterative Deepening Search, dan Uniform Cost Search, tidak menggunakan informasi tambahan tentang tujuan, hanya mengandalkan struktur graf.

Sebaliknya, pencarian dengan informasi menggunakan heuristik, seperti Best First Search dan A\*, untuk memandu pencarian menuju solusi optimal. Meskipun kurang umum untuk penelusuran graf dasar, ini relevan untuk masalah kompleks seperti pencarian jalur dalam graf berbobot.

### 2.2. Breadth First Search (BFS)



### Gambar 2.2.1 Ilustrasi BFS

(Sumber: [www.freelancinggig.com/](http://www.freelancinggig.com/))

BFS merupakan algoritma pencarian graf menggunakan prinsip queue (antrean). BFS menelusuri graf secara melebar dan mengunjungi semua simpul pada level saat ini sebelum pindah ke level berikutnya. BFS menggunakan antrian untuk menyimpan simpul yang akan dikunjungi. Simpul pertama yang dimasukkan ke antrean adalah simpul yang akan dikunjungi pertama kali dan simpul berikutnya akan mengikuti urutan masuk. BFS juga menjamin untuk menemukan jalur terpendek dalam graf yang tidak berbobot. BFS dapat digunakan jika kita ingin mencari solusi dengan jalur terpendek, komponen saling terhubung, atau menjelajahi struktur graf/data secara melebar.

Langkah algoritma BFS:

Misalkan ada sebuah simpul  $v$  dari graf  $G$ . Akan dilakukan traversal graf  $G$  dengan algoritma BFS dimulai dari simpul  $v$ . Skema algoritmanya adalah sebagai berikut:

1. Kunjungi simpul  $v$  dan tandai sebagai sudah dikunjungi.
2. Masukkan simpul  $v$  ke dalam antrian (queue).
3. Selama antrian tidak kosong:
  - Ambil simpul dari depan antrian (dequeue).
  - Kunjungi seluruh simpul yang bertetangga dengan simpul tersebut, yang belum dikunjungi.
  - Tandai simpul-simpul tetangga tersebut sebagai sudah dikunjungi dan masukkan ke dalam antrian.
4. Ulangi langkah ini hingga seluruh simpul pada graf  $G$  telah dikunjungi.

BFS
{ Melakukan traversal graf dengan algoritma pencarian BFS. Masukan: $v$ adalah simpul awal traversal Luaran: Seluruh simpul graf dijelajahi dengan algoritma pencarian BFS }
<b>Deklarasi</b>
w : simpul q : antrian
procedure BuatAntrian(input/output q : antrian) { Membuat antrian kosong }

```

procedure MasukAntrian(input/output q : antrian, input v : simpul) {Memasukkan
simpul v ke antrian q pada posisi belakang }
procedure HapusAntrian(input/output q : antrian, output v : simpul) { Menghapus
v dari kepala antrian q.}
function AntrianKosong(input q : antrian) → boolean { Mengecek apakah antrian q
kosong }

```

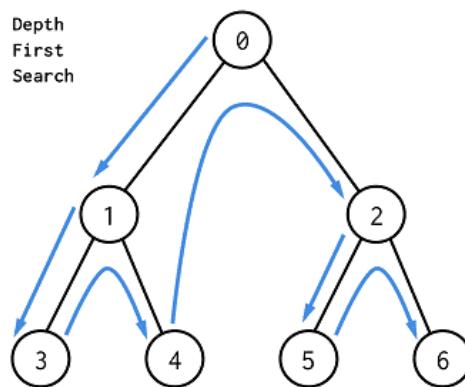
#### Algoritma

```

BuatAntrian(q)
write(v)
Kunjungi(v) ← true
MasukAntrian(q,v) { Kunjungi semua simpul graf selama antrian belum kosong }
while not AntrianKosong(q) do
    HapusAntrian(q,v)
    for (tiap simpul w yang bertetangga dengan simpul v) do
        if not dikunjungi(w) then
            Kunjungi(w)
            MasukAntrian(q,w)
        endif
    endfor
endwhile
{ AntrianKosong(q) }

```

### 2.3. Depth First Search (DFS)



Gambar 2.3.1 Ilustrasi DFS

(Sumber: [www.freelancinggig.com/](http://www.freelancinggig.com/))

DFS merupakan algoritma pencarian graf menggunakan prinsip stack (tumpukan). DFS menelusuri graf secara mendalam, menjelajahi satu cabang sebanyak mungkin sebelum kembali

ke cabang lain. DFS menggunakan tumpukan untuk menyimpan simpul yang akan dikunjungi. Simpul terakhir yang dimasukkan ke tumpukan adalah simpul yang akan dikunjungi pertama kali dan simpul berikutnya akan mengikuti urutan masuk. DFS sering menggunakan algoritma *backtracking*, yaitu kembali ke simpul sebelumnya jika tidak ada simpul lain yang bisa dikunjungi. DFS dapat digunakan jika kita ingin mencari solusi tunggal, komponen terhubung *biconnected*, atau menjelajahi struktur data secara mendalam.

Langkah algoritma DFS:

Misalkan ada sebuah simpul  $v$  dari graf  $G$ . Akan dilakukan traversal graf  $G$  dengan algoritma DFS dimulai dari simpul  $v$ . Skema algoritmanya adalah sebagai berikut:

1. Kunjungi simpul  $v$  dan tandai sebagai sudah dikunjungi.
2. Untuk setiap simpul tetangga dari  $v$  yang belum dikunjungi:
  - Lakukan DFS secara rekursif pada simpul tetangga tersebut.
3. Ulangi langkah ini hingga seluruh simpul yang terhubung dengan  $v$  telah dikunjungi.
4. Jika graf tidak terhubung, ulangi proses dari simpul lain yang belum dikunjungi hingga seluruh simpul pada graf  $G$  telah dikunjungi.

```
DFS
{
    Melakukan traversal graf dengan algoritma DFS rekursif.
    Masukan: v adalah simpul awal traversal
    Luaran: Semua simpul graf dijelajahi }

Deklarasi
procedure DFS(v : simpul)
w : simpul

{ Array atau struktur penanda kunjungan }
Kunjungi(v) : boolean

Algoritma
procedure DFS(v : simpul)
    if not Kunjungi(v) then
        write(v)
        Kunjungi(v) ← true
        for (tiap simpul w yang bertetangga dengan simpul v) do
            DFS(w)
        endfor
    endif
}
```

## **2.4. Aplikasi Web**

Dalam tugas besar ini, dikembangkan sebuah aplikasi web interaktif yang mengintegrasikan teknologi front-end dan back-end untuk menghadirkan pengalaman bermain game "Little Alchemy 2" yang cepat, responsif, dan intuitif. Aplikasi ini memungkinkan pengguna menggabungkan berbagai elemen dasar untuk menemukan elemen baru, meniru mekanisme permainan eksploratif seperti pada "Little Alchemy 2". React.js digunakan pada sisi front-end untuk membangun antarmuka pengguna yang dinamis dan efisien, sementara Golang digunakan pada sisi back-end untuk menangani logika kombinasi elemen, penyimpanan progres pengguna, dan penyajian data secara optimal. Untuk mendukung proses pengembangan dan deployment yang lebih stabil dan konsisten, aplikasi ini juga dikemas menggunakan Docker, yang memungkinkan setiap komponen dijalankan di dalam container terisolasi dengan lingkungan yang telah dikonfigurasi secara konsisten.

### **2.4.1. Next.js**

Next.js adalah framework open-source untuk membuat aplikasi web menggunakan bahasa pemrograman JavaScript. Sederhananya, Next.js memudahkan pengembangan aplikasi web dengan menyediakan sejumlah fitur dan fungsi yang dapat mempercepat proses pembuatan dan pengembangan. Next.js memberikan banyak manfaat bagi pengembang aplikasi web, termasuk kemampuan untuk membuat tampilan (UI) yang responsif dan SEO-friendly dengan cepat, mengelola state aplikasi dengan mudah, serta mengoptimalkan kinerja aplikasi web secara otomatis. Selain itu, Next.js juga memiliki dukungan untuk Server-side Rendering (SSR) dan Static Site Generation (SSG), yang memungkinkan aplikasi web untuk dihasilkan dan diakses dengan lebih cepat.

### **2.4.2. Golang**

Golang, atau Go, adalah bahasa pemrograman yang dikembangkan oleh Google dan dikenal karena sintaksnya yang sederhana, performa tinggi, serta efisiensi dalam menangani konkurensi. Dalam tugas besar ini, Golang digunakan di sisi backend untuk mengelola logika aplikasi, berinteraksi dengan database, serta melakukan integrasi dengan sistem lainnya. Bahasa ini sangat cocok untuk pengembangan aplikasi yang membutuhkan kecepatan dan skalabilitas, terutama pada lingkungan multi-core.

### **2.4.3. Docker**

Docker adalah platform *containerization* yang digunakan untuk membungkus aplikasi beserta seluruh dependensinya ke dalam satu unit terisolasi yang disebut container. Dalam proyek ini, Docker digunakan untuk melakukan deployment aplikasi secara konsisten di berbagai lingkungan, baik dalam pengembangan maupun produksi. Dengan Docker, komponen front-end (React.js) dan back-end (Golang) dapat dijalankan dalam container masing-masing tanpa konflik konfigurasi antar sistem. Ini mempermudah proses instalasi, testing, dan deployment karena setiap container memiliki lingkungan eksekusi tersendiri yang sudah terdefinisi. Selain itu, Docker juga mendukung skalabilitas dan portabilitas aplikasi secara efisien.

## BAB III – ANALISIS PEMECAHAN MASALAH

### 3.1. Langkah Pemecahan Masalah

#### 3.1.1. Pendefinisian Masalah

Masalah pencarian resep elemen-elemen dalam permainan Little Alchemy 2 berkaitan dengan mengkombinasikan berbagai elemen dasar untuk menghasilkan elemen yang baru. Elemen-elemen baru dapat dikombinasikan dengan elemen dasar atau elemen lainnya untuk menghasilkan elemen-elemen dengan tingkat yang lebih tinggi. Permasalahan komputasi pada tugas 2 ini berorientasi pada pencarian resep untuk suatu elemen target yang dapat dipilih oleh user.

#### 3.1.2. Perancangan Solusi

Kami menganalisis terdapat dua pendekatan yang dapat diterapkan untuk memecahkan permasalahan ini, yaitu *bottom-up* dan *top-down*.

Pendekatan bottom-up memulai pencarian dari elemen-elemen dasar (seperti *Earth*, *Fire*, *Air*, dan *Water*), lalu secara sistematis menggabungkan kombinasi yang mungkin untuk membangun elemen-elemen baru hingga mencapai elemen target. Pendekatan ini sesuai digunakan pada algoritma BFS, karena BFS secara alami mengeksplorasi level demi level dari elemen-elemen yang dapat dibuat, sehingga menyerupai proses pemain bereksperimen dari bawah menuju elemen kompleks di permainan. Namun, untuk metode multiple recipe BFS, digunakan pendekatan top-down karena sifatnya yang berusaha mencari variasi resep, sehingga pencarian akan lebih fokus pada penemuan resep dibandingkan dengan penemuan resep paling pendek.

Sementara itu, pendekatan top-down memulai pencarian dari elemen target, lalu secara bertahap mendekomposisi elemen tersebut ke dalam pasangan bahan pembentuknya hingga mencapai elemen dasar. Pendekatan ini kami gunakan dalam penerapan DFS, karena traversal secara mendalam dari target ke base element memungkinkan eksplorasi yang lebih efisien dan fokus pada jalur-jalur yang memang berpotensi menghasilkan target. Hal ini mencegah eksplorasi jalur-jalur tidak relevan yang tidak akan pernah menghasilkan target (seperti yang bisa terjadi pada bottom-up DFS), sehingga meningkatkan efisiensi pencarian, terutama saat menggunakan mode multi-path dengan batas jumlah solusi tertentu.

Dengan demikian, kami menerapkan:

- Algoritma BFS dengan pendekatan bottom-up, untuk menyimulasikan proses kombinasi elemen secara menyeluruh dan sistematis.
- Algoritma DFS dengan pendekatan top-down, untuk fokus mendalami jalur pembuatan elemen target secara lebih terarah dan efisien, terutama saat digunakan dengan iteratif yang mendukung deduplikasi jalur.

Pendekatan yang berbeda ini dipilih berdasarkan karakteristik masing-masing algoritma dan efektivitasnya dalam mengeksplorasi ruang pencarian resep pada permainan *Little Alchemy 2*.

### **3.1.3. Implementasi Solusi**

Setelah kami berhasil mendefinisikan masalah dan merancang solusi yang mungkin, kami memutuskan framework frontend dan backend yang digunakan pada aplikasi ini. Untuk frontend kami menggunakan framework React, sedangkan untuk backend API kami memilih net/http, *built-in Go package*. Kemudian, kami membuat desain software yang relatif mudah digunakan oleh user secara umum. Terdapat beberapa input yang dapat dipilih oleh user, yaitu elemen yang ingin dicari resepnya, algoritma yang ingin digunakan, dan banyak variasi resep yang ingin dicari.

### **3.1.3 Integrasi Backend dan Frontend**

Ketika tombol "Find Results" ditekan, frontend akan mengirimkan *HTTP request* ke backend API dengan fetch API. Data yang dikirim mencakup informasi elemen target, jenis algoritma, dan jumlah resep yang diinginkan. Di backend, endpoint khusus menerima dan memproses permintaan sesuai algoritma yang dipilih, memungkinkan sistem menghasilkan dan mengembalikan hasil pencarian yang sesuai dalam format *HTTP response*.

### **3.1.4 Pengoptimalan Program**

Proses pencarian dalam sistem ini mengimplementasikan teknik multi-*threading* menggunakan Go Routines di dalam Go. Teknik ini memungkinkan pencarian multiresep dilakukan secara paralel dan konkuren, sehingga beberapa pencarian dapat dijalankan bersamaan. Dengan demikian, efisiensi pemrosesan meningkat karena pencarian jalur dapat dieksekusi secara bersamaan, memanfaatkan sepenuhnya kemampuan banyak core CPU.

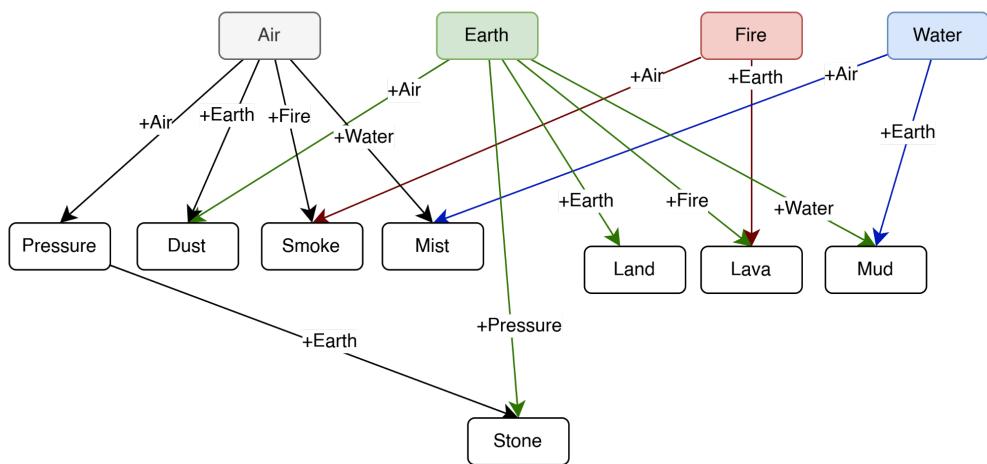
Selain itu, kami juga memutuskan untuk menggunakan representasi graf berbasis integer, di mana setiap elemen diberikan ID integer yang unik. Pendekatan ini membuat operasi, perbandingan, dan pencarian elemen dalam graf menjadi lebih

efisien. Dengan menggunakan ID integer sebagai pengidentifikasi, waktu akses dan pembaruan data menjadi lebih cepat, yang sangat mendukung efisiensi keseluruhan dalam proses pencarian. Struktur data ini diimplementasikan dengan *adjacency list* atau *hashmap* yang terindeks, memungkinkan traversal graf yang lebih cepat dan memungkinkan pencarian jalur dilakukan dengan lebih efektif.

### 3.1.5. Evaluasi dan Pengujian

Setelah program selesai dirancang, dilakukan pengujian terhadap program dengan mencoba beberapa kasus yang mencakup berbagai jalur resep dalam permainan Little Alchemy 2. Hasil pencarian jalur resep kemudian dibandingkan dengan referensi yang ada, seperti data yang tersedia di platform lain atau basis data resep yang sudah ada. Proses ini dilakukan untuk memastikan akurasi dan efisiensi dalam menemukan jalur resep yang tepat. Selain itu, program diuji dengan menggunakan teknik perbandingan dengan algoritma lain untuk memastikan bahwa pencarian dapat dilakukan dengan waktu yang cepat, meskipun dengan jumlah jalur yang banyak. Revisi dilakukan berdasarkan hasil pengujian, dan pengujian diulang berkali-kali untuk memastikan bahwa sistem dapat mencari jalur resep dalam waktu yang optimal dan sesuai dengan jumlah jalur yang diminta.

## 3.2. Pemetaan Masalah



Gambar 3.2.1 Contoh Sebagian Representasi Graf Little Alchemy 2

Sumber: Dokumentasi Penulis

Pada pencarian resep elemen dalam permainan Little Alchemy 2, semua kombinasi elemen telah ditentukan sebelumnya melalui graf berarah yang dibangun dari data resep (recipe.json). Graf ini bersifat statis, dengan semua elemen dan hubungan kombinasinya diketahui sejak awal. Meskipun demikian, eksplorasi elemen dilakukan secara bertahap selama pencarian. Berikut adalah pemetaan masalah dalam konteks graf statis:

- Akar: elemen dasar (Earth, Fire, Air, Water) untuk BFS; elemen target untuk DFS.
- Simpul: elemen-elemen yang bukan target atau elemen dasar selama pencarian.
- Daun: elemen target (BFS) atau elemen dasar (DFS).
- Ruang Solusi: kumpulan resep yang mencapai elemen target (satu untuk single path, beberapa untuk multi path)
- Ruang Status: seluruh elemen yang telah dieksplorasi dalam graf.
- Operator: mengecek apakah kombinasi elemen menghasilkan elemen target (BFS) atau apakah elemen dapat diuraikan ke elemen dasar (DFS).

### 3.2.1. Pemetaan Masalah menjadi Elemen Algoritma BFS

Elemen (umum):

- **Awal:** elemen dasar (Earth, Fire, Air, Water) sebagai root atau start node.
- **Akhir:** elemen target (misalnya Obsidian) yang dicari untuk menyimpan jalur resep.
- **Child dari Node:** kombinasi elemen yang dihasilkan dari node saat ini, diambil dari IndexedGraph.
- **Queue:** menyimpan elemen untuk dieksplorasi; dalam multi-path, menyimpan state dengan jalur lengkap.
- **Visited Map:** mencatat elemen yang sudah dikunjungi (seen atau reachable) untuk mencegah pengulangan
- **Parent Map:** mencatat hubungan parent-partner (prevIDs atau state.path) untuk merekonstruksi resep.

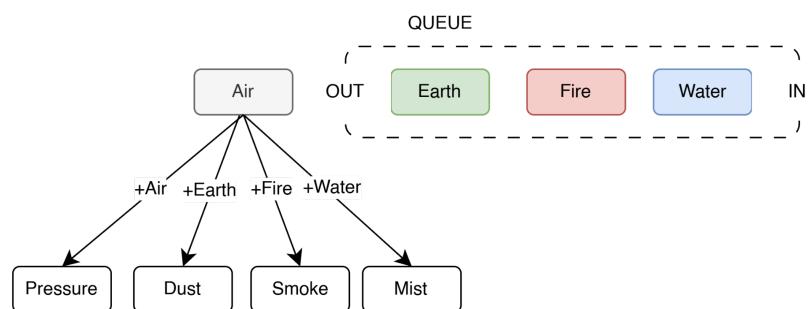
Proses untuk Single-Recipe Search:

1. **Inisialisasi:** masukkan elemen dasar ke queue dan tandai sebagai dikunjungi (seen).



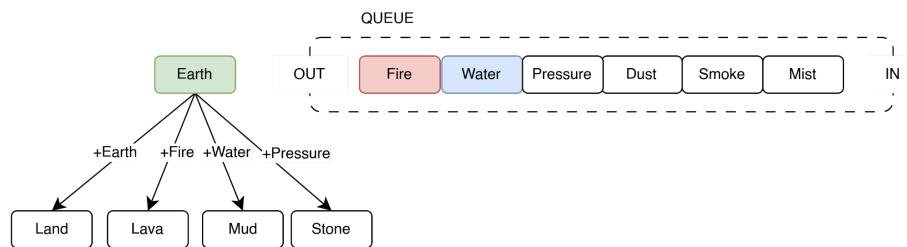
Sumber: Dokumentasi Penulis

2. **Eksplorasi dan penyimpanan:** Ambil elemen dari queue (misal, Air), cari tetangga (misal, Dust) yang merupakan hasil kombinasi dengan elemen lain yang sudah pernah dikunjungi atau ditemukan (misal, Earth) menggunakan IndexedGraph. Hasilkan produk (dalam kasus ini Dust) dan simpan dalam queue. Jika produk adalah target, hentikan pencarian.



Gambar 3.2.1.1 Ilustrasi *dequeue* elemen pertama (Air)

Sumber: Dokumentasi Penulis



Gambar 3.2.1.2 Ilustrasi *dequeue* elemen kedua (Earth)

Sumber: Dokumentasi Penulis

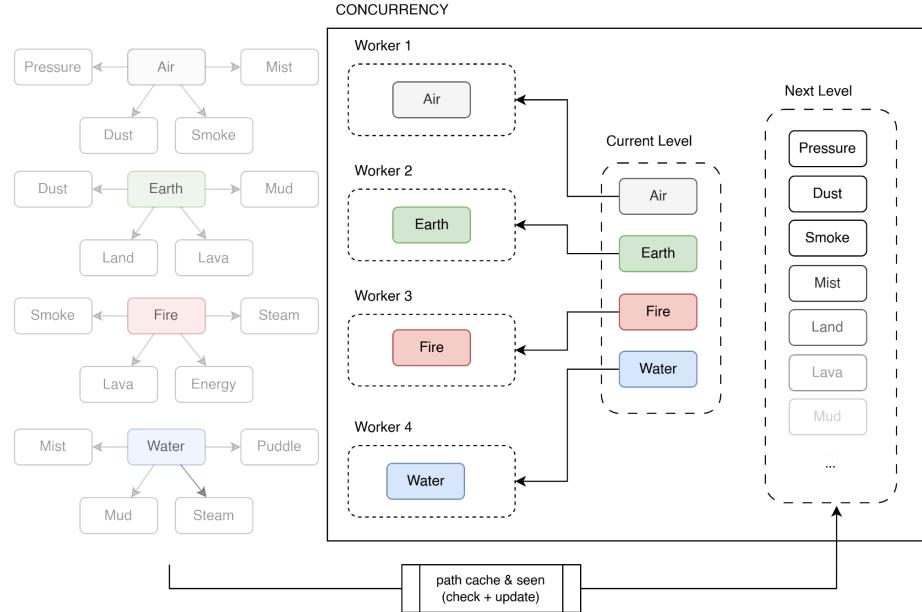
Catatan: perhatikan bahwa sebenarnya Air memiliki banyak tetangga lain, seperti Heat (Air + Energy). Namun, node Heat tidak diaktifkan karena program belum pernah mengunjungi node Energy. Walaupun demikian, pada akhirnya, program tetap akan mengekspan node Heat ketika node Energy telah diekspan sebelumnya. Perilaku ini akan

memastikan bahwa node tertentu (dari hasil 2 *reagent* spesifik) akan tetap dikunjungi meskipun dilewati sebelumnya oleh salah satu node *reagent*.

3. **Iterasi:** Proses berlanjut hingga target ditemukan atau queue kosong, mencatat hubungan parent-partner dalam prevIDs untuk membangun resep.

Proses untuk Multi-Recipe Search:

1. **Inisialisasi:** masukkan elemen dasar ke queue dengan state awal (jalur kosong), tandai sebagai dikunjungi.
2. **Eksplorasi dan penyimpanan:** Ambil state dari queue, cari tetangga yang merupakan hasil kombinasi dengan elemen lain yang sudah pernah dikunjungi atau ditemukan, dan tambahkan state baru dengan jalur diperbarui ke queue berikutnya. Jalur di-hash untuk deduplikasi.
3. **Multithreading:** bagi queue berdasarkan kedalaman ke worker (dibatasi oleh jumlah CPU). Setiap worker memproses subqueue-nya.
4. **Proses worker:** worker mengeksplorasi kombinasi, memeriksa target, dan menambahkan jalur unik ke hasil hingga batas recipe yang diinginkan user tercapai.



Gambar 3.2.1.3 Ilustrasi worker pada proses *multithreading BFS*  
Sumber: Dokumentasi Penulis

5. Proses berlanjut hingga semua jalur yang diinginkan tercapai.

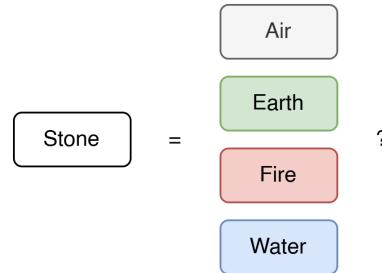
### 3.2.2. Pemetaan Masalah menjadi Elemen Algoritma DFS

Elemen:

- **Awal:** elemen target (misalnya, Obsidian) sebagai root atau start node.
- **Akhir:** elemen dasar (Earth, Fire, Air, Water) sebagai target akhir.
- **Child dari Node:** pasangan bahan yang menghasilkan elemen saat ini.
- **Stack:** Implisit melalui rekursi untuk Single-Recipe Search; eksplisit menyimpan elemen dan posisi anak untuk Multi-Recipe Search.
- **Visited Map:** mencatat elemen yang sudah dikunjungi untuk mencegah siklus.
- **Parent Map:** mencatat hubungan parent-partner untuk merekonstruksi resep.

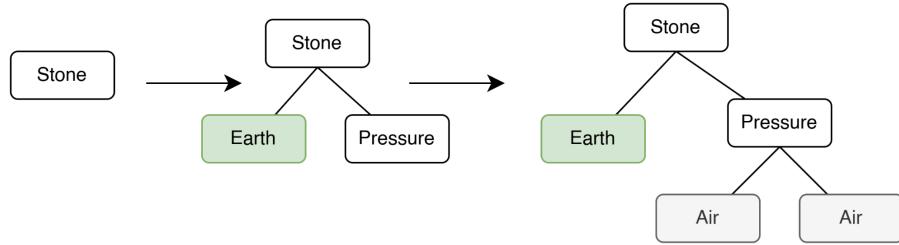
Proses untuk Single-Recipe Search:

1. **Pengecekan awal:** Periksa apakah elemen saat ini (mulai dari target) adalah elemen dasar dengan membandingkan dengan BaseElements. Jika ya, tandai sebagai dapat mencapai dasar dan hentikan (jarang terjadi karena target biasanya bukan elemen dasar). Jika tidak, lanjutkan pencarian.



Gambar 3.2.2.1 Pengecekan elemen target  
Sumber: Dokumentasi Penulis

2. **Eksplorasi dan penyimpanan:** uraikan target ke pasangan bahan (misal, Air + Pressure) secara rekursif. Jika kedua bahan mencapai elemen dasar, simpan resep dalam recipes.



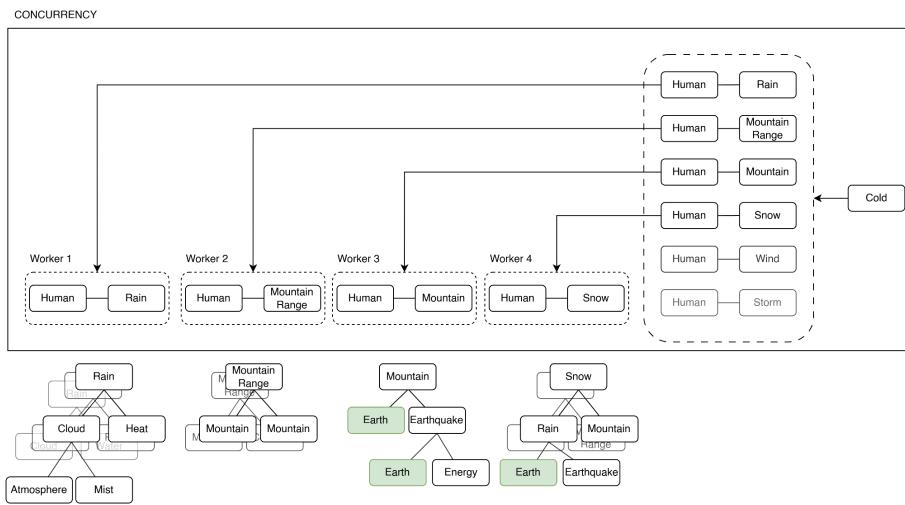
Gambar 3.2.2.2 Proses dekomposisi elemen target

Sumber: Dokumentasi Penulis

3. Proses berlanjut dengan memoization hingga resep ditemukan atau batas kedalaman tercapai.

Proses untuk Multi-Recipe Search:

1. **Pengecekan Awal:** Inisialisasi pencarian dari target dengan pasangan bahan dari revIdx sebagai awal worker. Setiap worker memeriksa apakah elemen awal (target) adalah elemen dasar sebelum memulai eksplorasi (jarang terjadi).
2. **Eksplorasi dan penyimpanan:** setiap worker menjalankan DFS berbasis stack, mengurai elemen ke bahan-bahan, dan membangun jalur. Jalur di-hash untuk deduplikasi
3. **Multithreading:** luncurkan worker per pasangan bahan (dibatasi jumlah CPU) untuk bereksplorasi secara independen.
4. **Proses worker:** worker memeriksa elemen bahan, menambahkan jalur unik ke hasil jika batas jumlah recipe belum tercapai, dan menghentikan proses saat selesai.

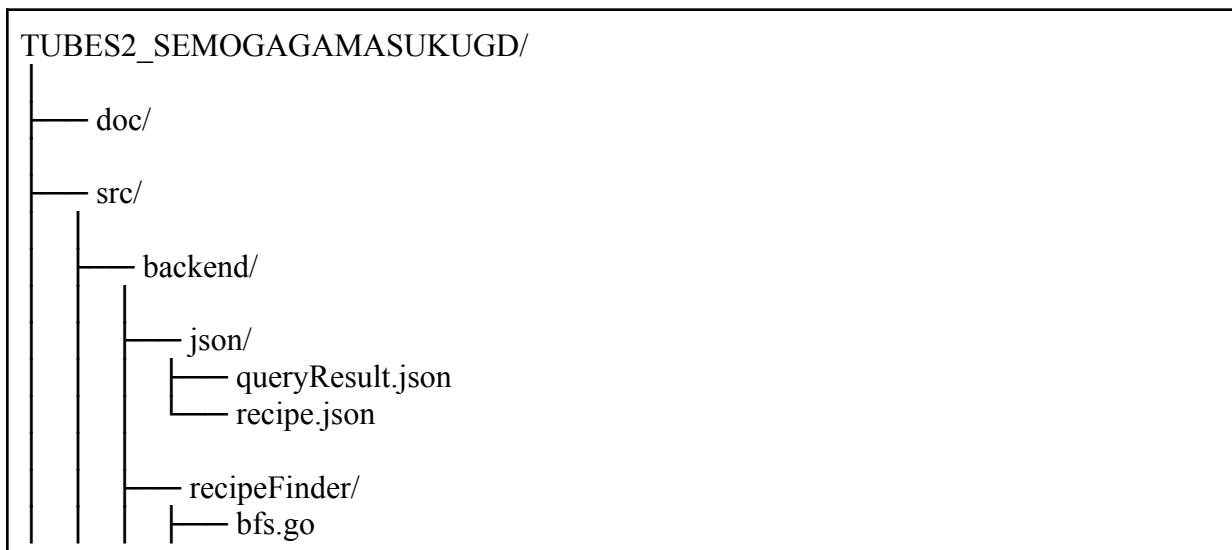


Gambar 3.2.2.3 Ilustrasi worker pada proses *multithreading DFS*  
Sumber: Dokumentasi Penulis

5. Proses berlanjut hingga jumlah recipe maksimum tercapai atau semua kemungkinan habis.

### 3.3. Arsitektur Aplikasi Web

Dalam pengembangan aplikasi web, kami memastikan struktur kode terorganisir dengan jelas melalui pembagian dua direktori utama yaitu untuk frontend dan backend.



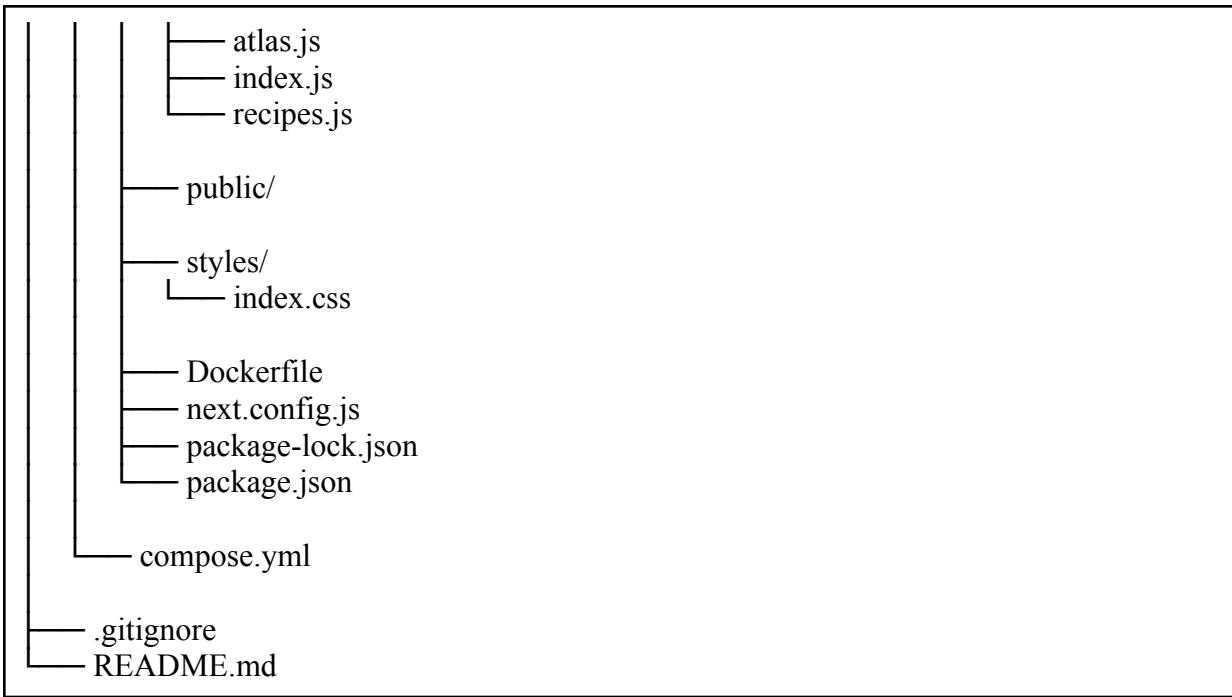
```
    └── bfs_multi.go
    └── dfs.go
    └── globals.go
    └── graph.go
    └── helpers.go
    └── scraper.go
    └── tree.go
    └── unify.go

    └── svgs/
        └── 1/
        └── 2/
        └── 3/
        └── 4/
        └── 5/
        └── 6/
        └── 7/
        └── 8/
        └── 9/
        └── 10/
        └── 11/
        └── 12/
        └── 13/
        └── 14/
        └── 15/
        └── starting/
        └── special/

    └── backend
    └── Dockerfile
    └── go.mod
    └── go.sum
    └── main.go

    └── frontend/
        └── .next/
            └── components/
                └── LiveSearchVisualizer.jsx
                └── RecipeTree.jsx
                └── SearchForm.jsx
                └── SearchTreeView.jsx

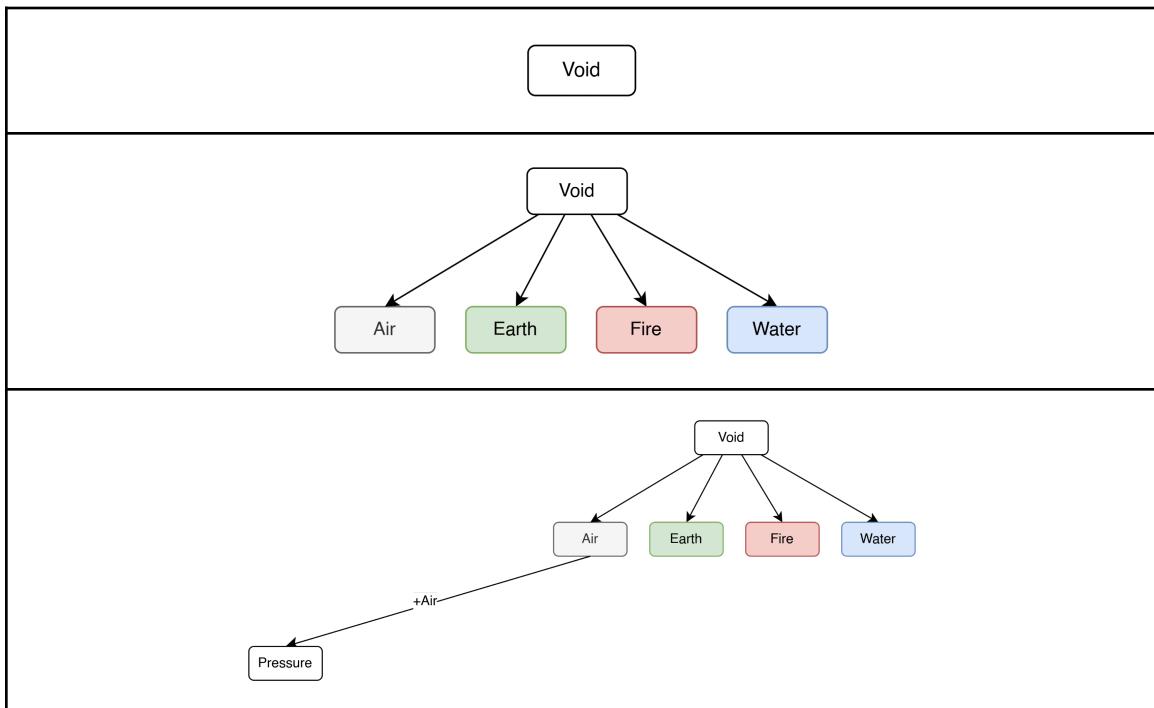
            └── pages/
```

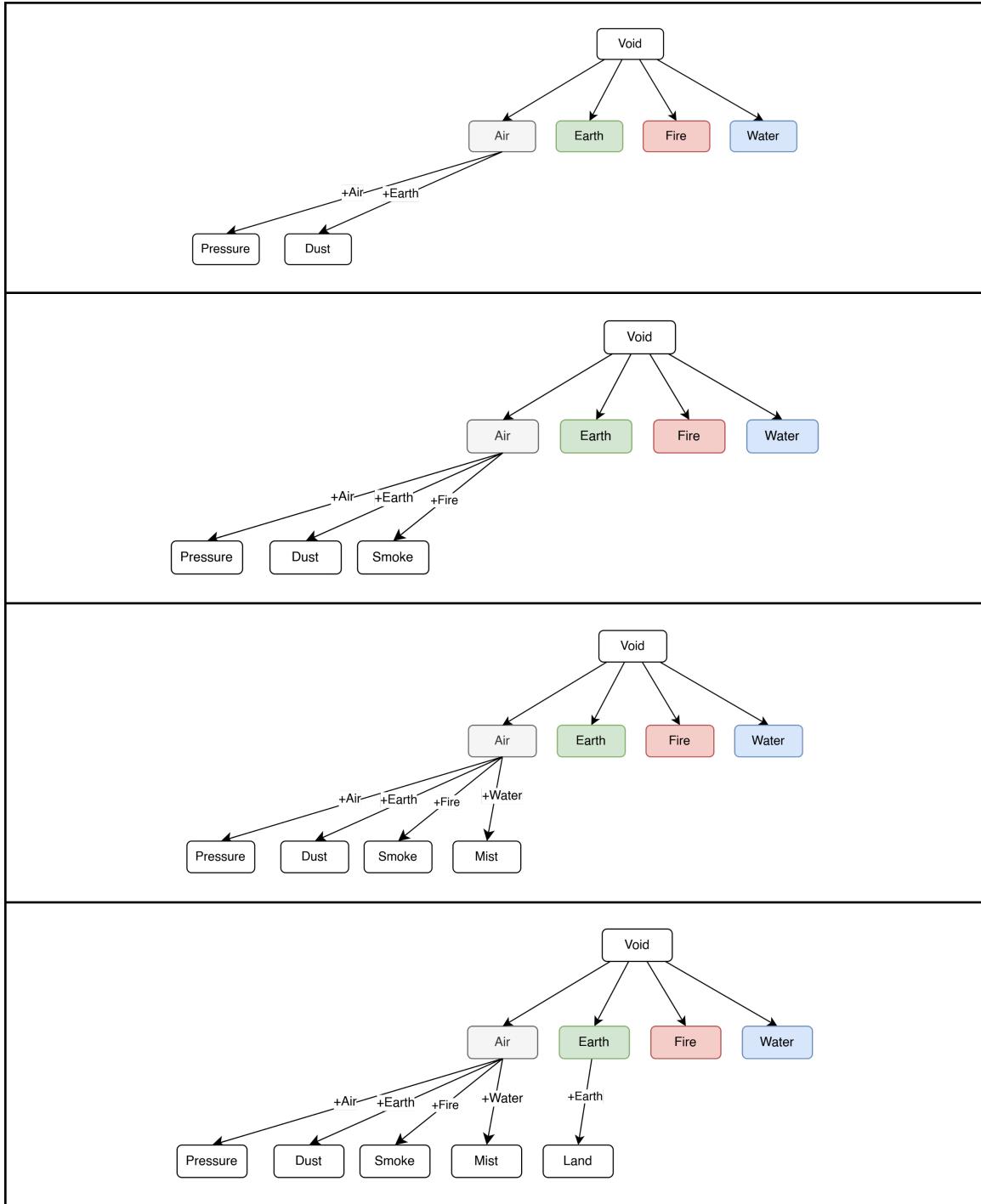


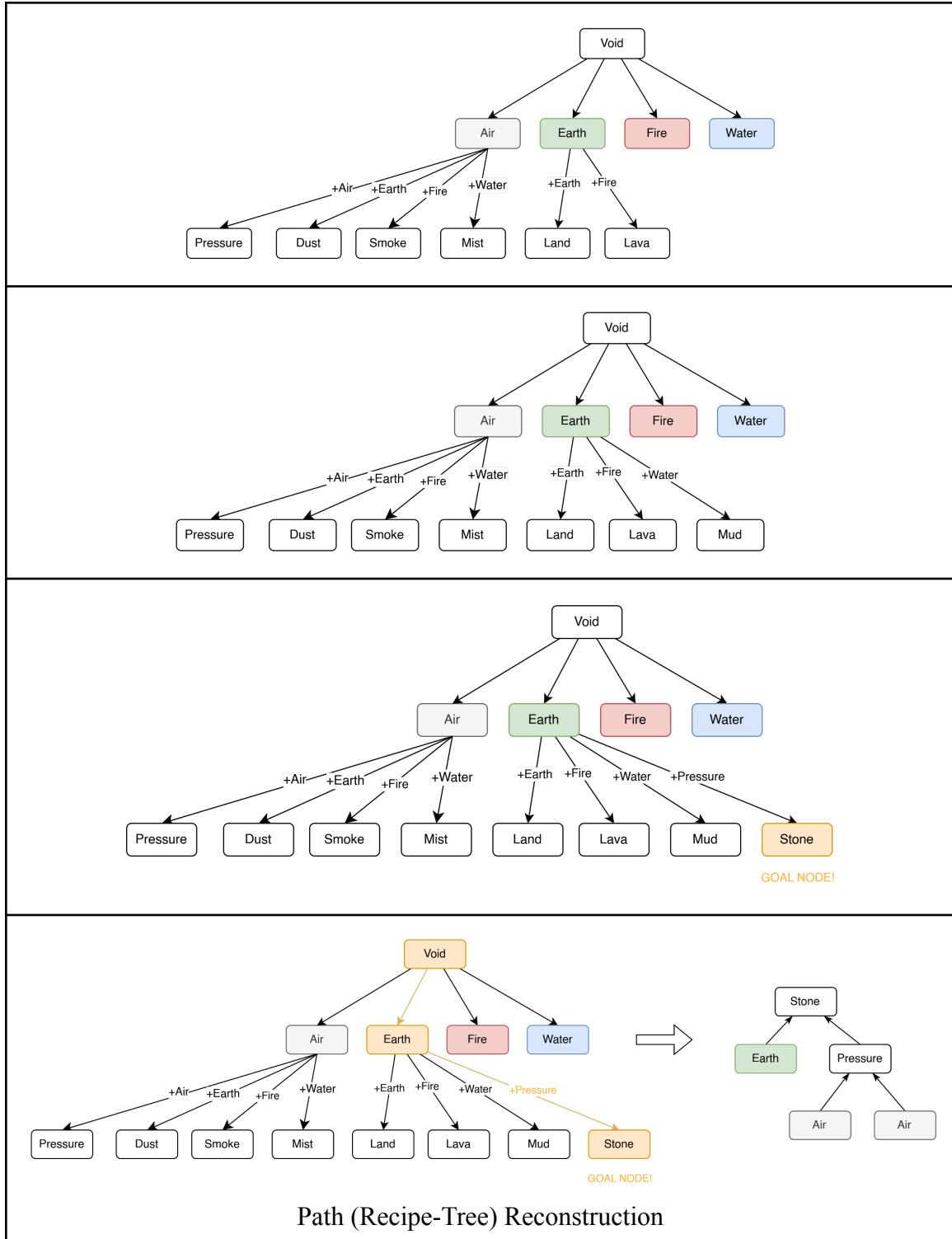
### 3.4. Ilustrasi Kasus

Berikut merupakan ilustrasi proses pencarian BFS dan DFS dengan elemen target Stone.

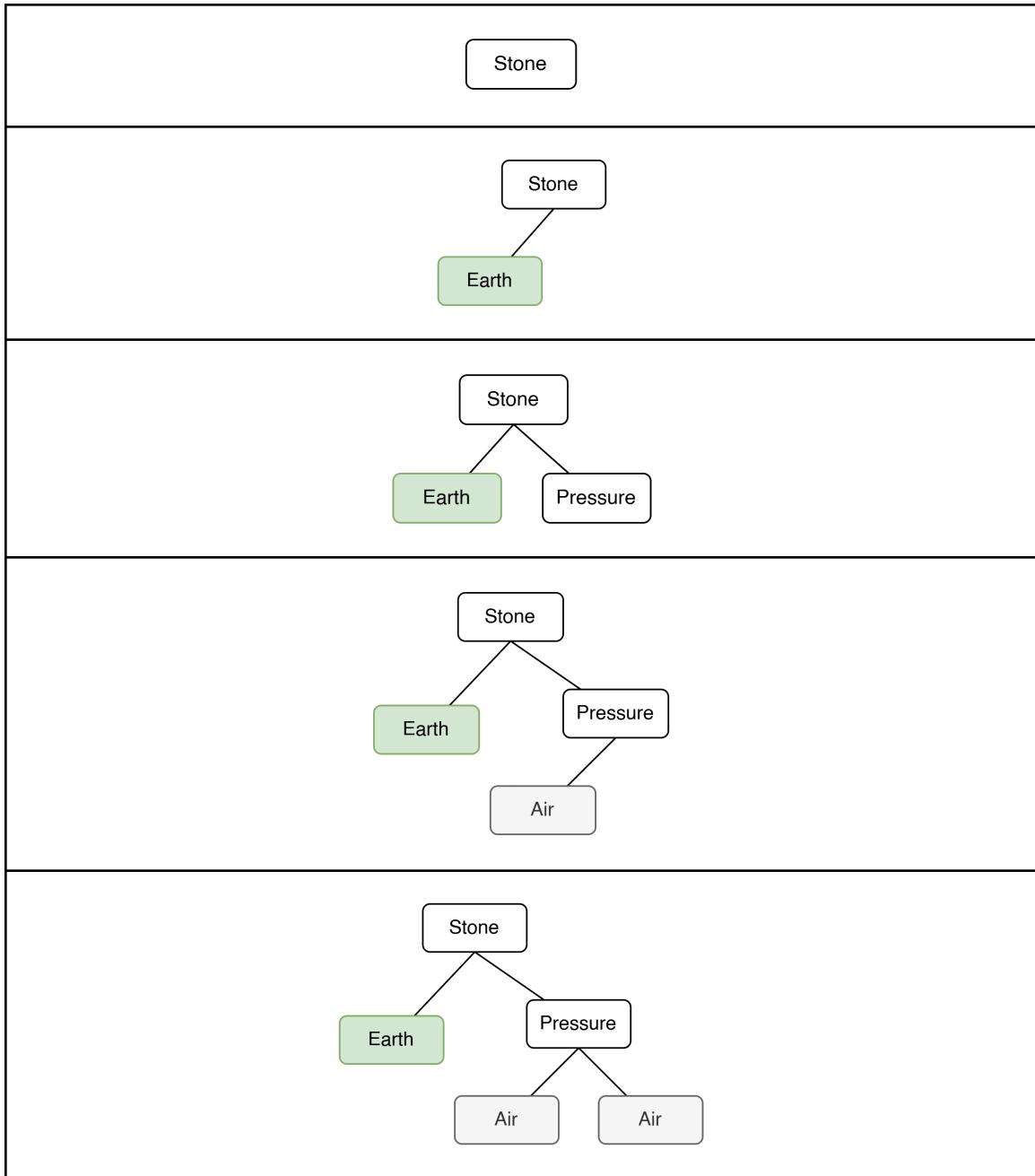
#### 3.4.1. BFS







### 3.4.2. DFS



## BAB IV – IMPLEMENTASI DAN PENGUJIAN

### 4.1. Spesifikasi Teknis dan Struktur Data Program

#### a. Struktur Data Program

##### i. Umum



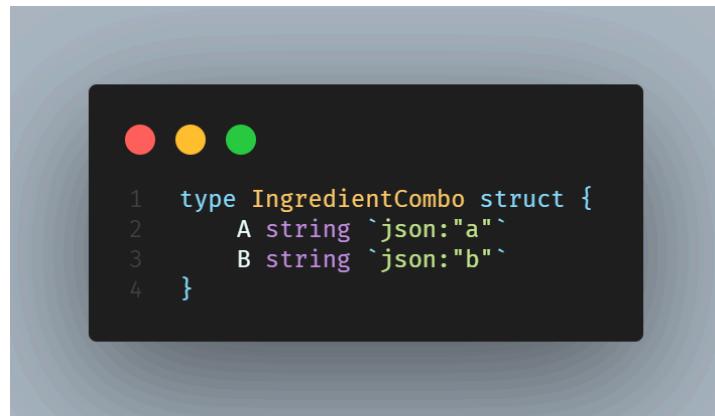
```
1 type Neighbor struct {
2     Partner string
3     Product string
4 }
5
6 type Graph map[string][]Neighbor
```

Graph merupakan map <string: slice of Neighbor> yang dipakai untuk menyimpan informasi bahan-bahan sebuah elemen

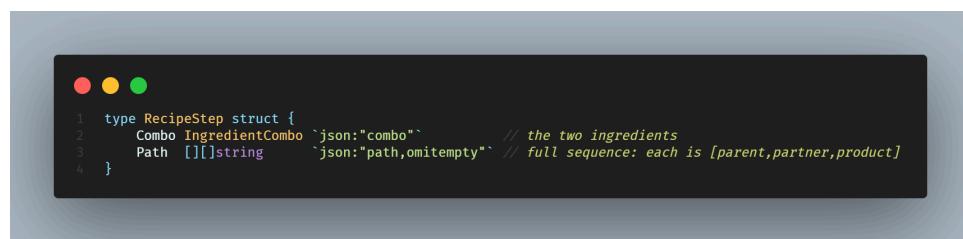


```
1 type IndexedGraph struct {
2     NameToID map[string]int           // Maps element names to their ID
3     IDToName map[int]string          // Reverse mapping for reconstruction
4     Edges    map[int][]IndexedNeighbor // Adjacency list using IDs
5 }
```

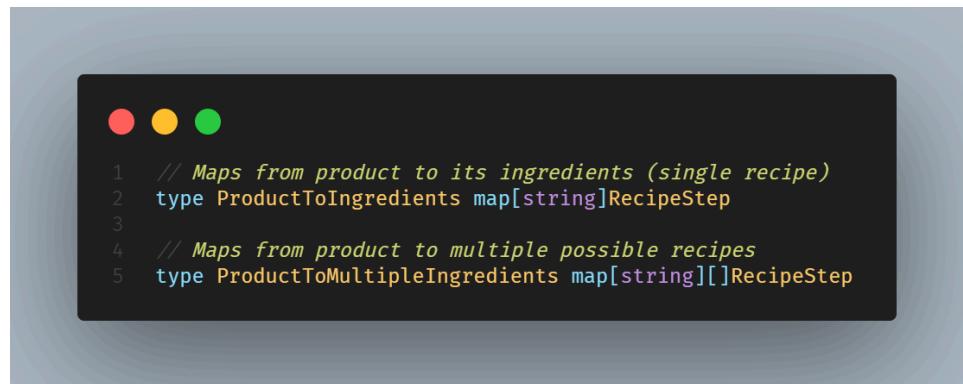
IndexedGraph merupakan versi Graph yang dapat dioperasikan menggunakan index berupa int untuk efisiensi map lookup.



IngredientCombo merupakan struct yang menyimpan kemungkinan dua elemen sebagai suatu resep.



RecipeStep merupakan struct yang menyimpan IngredientCombo beserta Path, yaitu jalur traversal hingga IngredientCombo saat itu.



ProductToIngredients adalah sebuah map dari suatu elemen (string) ke RecipeStep

ProductToMultipleIngredients adalah sebuah map dari suatu elemen (string) ke slice of RecipeStep



```
1 // ===== GLOBALS =====
2 // Global indexed graph accessible throughout the package
3 var GlobalIndexedGraph IndexedGraph
4
5 // Global variable to store the catalog
6 var GlobalCatalog Catalog
7
8 var revIdx revIndex
```

GlobalIndexedGraph adalah variable IndexedGraph global yang dipakai untuk seluruh algoritma.

GlobalCatalog adalah variabel Catalog global yang menyimpan informasi tentang semua elemen beserta resepnya.

revIdx adalah variabel revIndex global yang menyimpan informasi elemen bahan

## ii. BFS

```
type SearchStep struct {
    CurrentID      int          `json:"current_id"`
    CurrentName    string       `json:"current"`
    QueueIDs       []int        `json:"queue_ids"`
    QueueNames     []string     `json:"queue"`
    SeenIDs        []int        `json:"seen_ids"`
    SeenNames      []string     `json:"seen"`
    DiscoveredEdges map[int]struct{ParentID, PartnerID int} `json:"discovered_ids"`
    DiscoveredNames map[string]struct{A, B string}           `json:"discovered"`
    StepNumber     int          `json:"step"`
    FoundTarget    bool         `json:"found_target"`
}
```

SearchStep merupakan sebuah struct yang akan dipakai untuk visualisasi langkah-langkah traversal graf hingga target ditemukan.

```

type state struct {
    elem int
    path [][]int
    depth int
}

```

### iii. DFS

```

type pair struct{ a, b int } // Represents an ingredient pair (a,b)
type revIndex map[int][]pair // Maps product ID to all ingredient pairs

```

#### b. Fungsi dan Prosedur yang dibangun

bfs.go	
Fungsi dan Prosedur	Penjelasan
<pre> func IndexdBFSBuild(targetName string, graph IndexedGraph) (ProductToIngredients, []SearchStep, int) </pre>	Mencari satu resep dari starting elements ke targetName menggunakan BFS. Akan mengeluarkan hasil pencarian resep berupa ProductToIngredients, history langkah BFS berupa [ ]SearchStep, dan jumlah node yang dikunjungi dalam int. Function ini berfungsi untuk mencatat setiap step dan menghentikan pencarian jika targetName (target elemen) ditemukan
<pre> func findKthPathIndexed(targetID, skip int, g IndexedGraph) (RecipeStep, int) </pre>	Mencari jalur ke-(skip + 1) menuju elemen targetID secara deterministik. Function ini menggunakan BFS level-based paralel, men-sort neighbor untuk determinisme, dan menghindari jalur duplicate via hash
<pre> func RangePathsIndexed(targetID int, start, limit int, g IndexedGraph) ([]RecipeStep, int) </pre>	Mengembalikan limit jalur unik ke elemen targetID, dimulai dari indeks start. Jika start = 0, akan memanfaatkan IndexdBFSBuild untuk efisiensi, sisanya menggunakan

	findAdditionalPaths
<pre>func findAdditionalPaths(targetID, start, limit int, g IndexedGraph) ([]RecipeStep, int)</pre>	Membantu RangePathIndexed dengan menjalankan findKthPathIndexed beberapa kali

dfs.go	
Fungsi dan Prosedur	Penjelasan
<pre>func BuildReverseIndex(g IndexedGraph)</pre>	Membangun index terbalik (revIdx) dari produk ke pasangan bahan penyusunnya dan di-sort berdasarkan tingkat kompleksitas tier
<pre>func findPathToBaseCnt</pre>	DFS rekursif untuk mencari jalur ke starting elements. Function ini dapat mencegah siklus, menggunakan cache (canReachBaseCache), memperhatikan batas kedalaman, dan mencatat langkah-langkah ke dalam recipes
<pre>func DFSSBuildTargetToBase(target string, g IndexedGraph) (ProductToIngredients, int)</pre>	Mencari satu jalur dari target ke starting elements menggunakan DFS rekursif (findPathToBaseCnt)
<pre>func hashPath(p [][]int) uint64</pre>	Membuat hash 64-bit untuk jalur DFS
<pre>func RangeDFSPaths(target string, maxPaths int, g IndexedGraph) ([]RecipeStep, int)</pre>	Mencari beberapa jalur unik ke target menggunakan DFS paralel berbasis stack. Function ini menggunakan hashPath untuk menghindari duplikasi, menggunakan goroutine per psangan root, dan bounded dengan runtime.NumCPU() dan sync

graph.go	
Fungsi dan prosedur	Penjelasan

<pre>func BuildGraphFromCatalog(cat Catalog) Graph</pre>	Membangun graf tak berarah dari data recipe untuk digunakan dalam pencarian resep dengan BFS dan DFS. Graf ini merepresentasikan hubungan antara elemen dan produk yang dihasilkan dari kombinasi dua elemen
<pre>func BuildIndexedGraph(cat Catalog) IndexedGraph</pre>	Membuat graph yang di-optimasi dari Catalog ke bentuk IndexedGraph. Function ini memberi ID untuk semua element (starting element terlebih dahulu), lalu membangun sisi graph berdasarkan recipe
<pre>func (g *IndexedGraph) GetBaseElementIDs() []int</pre>	Mengembalikan ID untuk semua starting element
<pre>func InitElementTiers(cat Catalog)</pre>	Menginisialisasi elementTierCache, yaitu pemetaan nama elemen ke tier recipe dari Catalog
<pre>func getElementTier(element string) int</pre>	Mengembalikan nilai tier dari suatu elemen berdasarkan cache

helpers.go	
Fungsi dan prosedur	Penjelasan
<pre>func hashPath(p [][]int) uint64 {</pre>	Membuat hash string dari path untuk menghindari duplikasi
<pre>func buildRecipeStepFromPath(path [][]int, targetID int, g IndexedGraph) RecipeStep</pre>	Mengubah path numerik menjadi path nama (string) dalam RecipeStep
<pre>func min(a, b int) int</pre>	Mengembalikan nilai terkecil antara a dan b dan menjaga urutan konsisten untuk pasangan elemen
<pre>func max(a, b int) int</pre>	Mengembalikan nilai terbesar antara a dan b dan menjaga urutan konsisten untuk pasangan elemen

<pre>func extractPathFromRecipes(targetID int, recipes ProductToIngredients, g IndexedGraph) RecipeStep</pre>	Membentuk RecipeStep dari hasil ProductToIngredients
<pre>func mapKeysToSlice(m map[int]bool) []int</pre>	Mengambil semua key dari map[int] bool dan mengembalikannya ke dalam bentuk slice [ ]int
<pre>func mapKeysToNameSlice(m map[int]bool, g IndexedGraph) []string</pre>	Sama seperti mapKeysToSlice, tetapi hasilnya adalah nama-nama elemen, bukan dalam bentuk ID
<pre>func queueToSlice(q *list.List) []int</pre>	Mengubah isi queue menjadi slice [ ]int dan menyimpan state dari queue dalam bentuk slice agar mudah di-encode
<pre>func queueToNameSlice(q *list.List, g IndexedGraph) []string</pre>	Sama seperti queueToSlice tetapi mengubah ID menjadi nama elemen dengan bantuan graf g
<pre>func copyMap(m map[int]struct{ ParentID, PartnerID int }) map[int]struct{ ParentID, PartnerID int }</pre>	Membuat deep copy dari map yang memetakan produk ke parent dan partner nya. Function ini dibutuhkan untuk menyimpan state BFS pada setiap langkah tanpa terganggu oleh mutasi di iterasi berikutnya
<pre>func prevIDsToNames(m map[int]struct{ ParentID, PartnerID int }, g IndexedGraph) map[string]struct{ A, B string }</pre>	Mengubah map berbasis ID menjadi map berbasis nama elemen
<pre>func findIngredientsFor(productID int, g IndexedGraph) [][]int</pre>	Mengembalikan semua pasangan elemen yang bisa membuat productID

scraper.go	
Fungsi dan prosedur	Penjelasan
<pre>func ScrapeAll() (Catalog, error)</pre>	Melakukan web <i>scraping</i> dari wiki Little Alchemy 2, yang terdiri dari mengambil

	semua elemen, gambar SVG, dan resep kombinasinya. Kemudian mengelompokkan elemen berdasarkan tier. Hasilnya berupa Catalog dan disimpan ke dalam JSON
<pre>func cleanTierName(raw string) string</pre>	Merubah format nama tier dari format wiki dari Tier 1 elements menjadi 1
<pre>func downloadSVG(url, dest string)</pre>	Download gambar SVG dari URL dan menyimpannya secara lokal

tree.go	
Fungsi dan prosedur	Penjelasan
<pre>func BuildTrees(target string, pathPrev map[string][]RecipeStep) []*RecipeNode</pre>	Membangun semua recipe tree untuk elemen target berdasarkan jalur dari RecipeStep.path yang telah ditemukan pathPrev. Lalu mengubah setiap jalur menjadi struktur pohon dengan memanggil function BuildTree
<pre>func isBaseElement(name string) bool</pre>	Mengecek apakah name adalah salah satu starting elements
<pre>func BuildTree(name string, prev ProductToIngredients) *RecipeNode</pre>	Membangun satu recipe tree untuk elemen name secara rekursif berdasarkan mapping prev dari produk ke elemen-elemennya
<pre>func buildTreeRec(     name string,     prev ProductToIngredients,     visited map[string]bool,     depth int, ) *RecipeNode</pre>	Membuat node resep dari elemen name. Jika name adalah starting elements, sudah pernah dikunjungi, atau terlalu dalam maka berhenti. Jika ada info resep di prev, maka buat subtree dari 2 elemen penyusunnya. Jika tidak ada, lakukan fallback dengan BFS untuk membangun subtree

## 4.2. Implementasi program

### a. [bfs.go](#)

```

package recipeFinder

import (
    "container/list"
    "context"
    "fmt"
    "runtime"
    "sort"
    "sync"
    "time"
)

type SearchStep struct {
    CurrentID      int
    `json:"current_id"`
    CurrentName    string
    `json:"current"`
    QueueIDs       []int
    `json:"queue_ids"`
    QueueNames     []string
    `json:"queue"`
    SeenIDs        []int
    `json:"seen_ids"`
    SeenNames      []string
    `json:"seen"`
    DiscoveredEdges map[int]struct{ ParentID, PartnerID int }
    `json:"discovered_ids"`
    DiscoveredNames map[string]struct{ A, B string }
    `json:"discovered"`
    StepNumber      int
    `json:"step"`
    FoundTarget     bool
    `json:"found_target"`
}

/*
-----  

-----  

Single-recipe BFS

```

```

/*
func IndexedBFSBuild(targetName string, graph IndexedGraph)
(ProductToIngredients, []SearchStep, int) {
    targetID := graph.NameToID[targetName]

    queue := list.New()
    seen := make(map[int]bool)

    searchSteps := []SearchStep{}

    for _, baseName := range BaseElements {
        baseID := graph.NameToID[baseName]
        queue.PushBack(baseID)
        seen[baseID] = true
    }

    searchSteps = append(searchSteps, SearchStep{
        CurrentID:      -1,
        CurrentName:    "",
        QueueIDs:       queueToSlice(queue),
        QueueNames:     queueToNameSlice(queue, graph),
        SeenIDs:        mapKeysToSlice(seen),
        SeenNames:      mapKeysToNameSlice(seen, graph),
        DiscoveredEdges: make(map[int]struct{ ParentID, PartnerID
int }),
        DiscoveredNames: make(map[string]struct{ A, B string }),
        StepNumber:      0,
        FoundTarget:    false,
    })

    // Track parents using integer IDs
    prevIDs := make(map[int]struct{ ParentID, PartnerID int })
    nodes := 0
    for queue.Len() > 0 {
        curID := queue.Remove(queue.Front()).(int)
        nodes++

        searchSteps = append(searchSteps, SearchStep{
            CurrentID:      curID,

```

```

    CurrentName:      graph.IDToName[curID],
    QueueIDs:         queueToSlice(queue),
    QueueNames:       queueToNameSlice(queue, graph),
    SeenIDs:          mapKeysToSlice(seen),
    SeenNames:        mapKeysToNameSlice(seen, graph),
    DiscoveredEdges: copyMap(prevIDs), // Need a deep copy
    DiscoveredNames: prevIDsToNames(prevIDs, graph),
    StepNumber:       nodes,
    FoundTarget:      curID == targetID,
  })

  if curID == targetID {
    break
  }

  for _, neighbor := range graph.Edges[curID] {
    partnerID := neighbor.PartnerID
    productID := neighbor.ProductID

    if seen[partnerID] && !seen[productID] {
      // We found a new product - record path
      seen[productID] = true
      prevIDs[productID] = struct{ ParentID, PartnerID
int }{
        ParentID: curID,
        PartnerID: partnerID,
      }

      // If this is the target, we can stop immediately
      if productID == targetID {
        // Create a copy of prevIDs that includes the
target we just found
        finalDiscoveredEdges := copyMap(prevIDs)
        finalDiscoveredEdges[productID] = struct{
ParentID, PartnerID int }{
          ParentID: curID,
          PartnerID: partnerID,
        }
      }
    }
  }
}

```

```

        // Convert to names
        finalDiscoveredNames :=
prevIDsToNames(finalDiscoveredEdges, graph)

        // Create final search step with target found
searchSteps = append(searchSteps, SearchStep{
    CurrentID:           productID, // Use target
ID as current
    CurrentName:
graph.IDToName[productID], // Show target as current element
    QueueIDs:            queueToSlice(queue),
    QueueNames:          queueToNameSlice(queue,
graph),
    SeenIDs:             mapKeysToSlice(seen),
    SeenNames:           mapKeysToNameSlice(seen,
graph),
    DiscoveredEdges:     finalDiscoveredEdges, //
Include target
    DiscoveredNames:     finalDiscoveredNames, //
Include target
    StepNumber:          nodes + 1,
    FoundTarget:         true,
})
}

// Break out of both loops
goto TargetFound
}

queue.PushBack(productID)
}
}
}

TargetFound:
// Convert integer results to ProductToIngredients
recipes := make(ProductToIngredients)
for productID, info := range prevIDs {
    productName := graph.IDToName[productID]
    parentName := graph.IDToName[info.ParentID]

```

```

partnerName := graph.IDToName[info.PartnerID]

recipes[productName] = RecipeStep{
    Combo: IngredientCombo{
        A: parentName,
        B: partnerName,
    },
}
}

return recipes, searchSteps, nodes
}

/*
-----
-----
Path (skip+1) (distinct)
*/
// findKthPathIndexed finds the (skip+1)-th distinct path to
targetID using a level-based parallel BFS,
// with deterministic ordering: sorted neighbors and stable
bounding.
func findKthPathIndexed(targetID, skip int, g IndexedGraph)
(RecipeStep, int) {
    type state struct {
        elem  int
        path  [][]int
        depth int
    }

    // pre-sort neighbor edges for deterministic order
    for u := range g.Edges {
        sort.Slice(g.Edges[u], func(i, j int) bool {
            a, b := g.Edges[u][i], g.Edges[u][j]
            if a.PartnerID != b.PartnerID {
                return a.PartnerID < b.PartnerID
            }
            return a.ProductID < b.ProductID
        })
    }
}

```

```

}

// cache for seen paths (thread-safe)
const cacheCapacity = 10000
type entry struct{ key string }
cacheList := make([]entry, 0, cacheCapacity)
cacheMap := make(map[string]struct{})
var cacheMu sync.Mutex
evict := func() {
    if len(cacheList) > cacheCapacity {
        old := cacheList[0]
        cacheList = cacheList[1:]
        delete(cacheMap, old.key)
    }
}
checkAndAdd := func(sig string) bool {
    cacheMu.Lock()
    defer cacheMu.Unlock()
    if _, ok := cacheMap[sig]; ok {
        return false
    }
    cacheMap[sig] = struct{}{}
    cacheList = append(cacheList, entry{key: sig})
    evict()
    return true
}

// Track product revisits for diversity
productVisits := make(map[int]int)
const maxProductVisits = 5 // Allow more diverse paths for
each product

// Cache product-to-recipe mappings to ensure diversity
productRecipes := make(map[int]map[string]bool)

// helper to copy and extend a path
appendCopyPath := func(cur [][]int, a, b, c int) [][]int {
    np := make([][]int, len(cur)+1)
    copy(np, cur)
    np[len(cur)] = append(np[len(cur)], a, b, c)
    return np
}

```

```

        np[len(cur)] = []int{min(a, b), max(a, b), c}
        return np
    }

// initialization
reachable := make(map[int]bool)
var currLevel []state
for _, b := range BaseElements {
    id := g.NameToID[b]
    reachable[id] = true
    currLevel = append(currLevel, state{elem: id, depth: 0})
}

nodes := 0
hits := 0
const maxDepth = 30
maxWorkers := runtime.NumCPU()
const maxLevelSize = 10000

ctx, cancel := context.WithTimeout(context.Background(),
10*time.Second) // Increased timeout
defer cancel()

for depth := 0; depth <= maxDepth && len(currLevel) > 0;
depth++ {
    select {
    case <-ctx.Done():
        return RecipeStep{}, nodes
    default:
    }
    var nextLevel []state
    var mu sync.Mutex
    var wg sync.WaitGroup
    sem := make(chan struct{}, maxWorkers)

    for _, st := range currLevel {
        nodes++
        if st.elem == targetID {
            hits++

```

```

        if hits-1 == skip {
            return buildRecipeStepFromPath(st.path,
targetID, g), nodes
        }
        continue
    }
wg.Add(1)
sem <- struct{}{}
go func(st state) {
    defer wg.Done()
    defer func() { <-sem }()
    for _, r := range g.Edges[st.elem] {
        select {
        case <-ctx.Done():
            return
        default:
        }
    }

    partnerID := r.PartnerID
    productID := r.ProductID

    mu.Lock()
    reached := reachable[partnerID]
    revisitCount := productVisits[productID]

    // Initialize product recipe tracking if
needed
    if productRecipes[productID] == nil {
        productRecipes[productID] =
make(map[string]bool)
    }

    // Generate recipe signature
    recipeKey := fmt.Sprintf("%d+%d", min(st.elem,
partnerID), max(st.elem, partnerID))
    seenRecipe :=
productRecipes[productID][recipeKey]
    mu.Unlock()
}

```

```

        // Skip if partner is not reachable
        if !reached {
            continue
        }

        // Allow exploring even if product was seen
before, under conditions:
        // 1. If we haven't exceeded max visits for
this product
        // 2. If we haven't seen this specific recipe
for this product
        shouldExplore := !reachable[productID] ||
                        (!seenRecipe && revisitCount <
maxProductVisits)

        if !shouldExplore {
            continue
        }

        // Create extended path
        np := appendCopyPath(st.path, st.elem,
partnerID, productID)

        // Check path canonicalization to avoid
duplicates
        sig := fmt.Sprintf("%d", hashPath(np))
        if !checkAndAdd(sig) {
            continue
        }

        mu.Lock()
        // Track this recipe for this product
        productRecipes[productID][recipeKey] = true

        // If product was already reachable, increment
visit count
        if reachable[productID] {
            productVisits[productID]++
        }
    }
}

```

```

        // Add to next level
        nextLevel = append(nextLevel, state{elem:
productID, path: np, depth: st.depth + 1})
        mu.Unlock()
    }
} (st)
}
wg.Wait()

// sort nextLevel for stability before bounding
sort.Slice(nextLevel, func(i, j int) bool {
    if nextLevel[i].elem != nextLevel[j].elem {
        return nextLevel[i].elem < nextLevel[j].elem
    }
    return len(nextLevel[i].path) < len(nextLevel[j].path)
})

if len(nextLevel) > maxLevelSize {
    nextLevel = nextLevel[:maxLevelSize]
}

// update reachable
for _, st := range nextLevel {
    reachable[st.elem] = true
}
currLevel = nextLevel
}
return RecipeStep{}, nodes
}

```

b. [dfs.go](#)

```

package recipeFinder

import (

```

```

    "context"
    "runtime"
    "sort"
    "sync"
    "sync/atomic"
)

/*
Reverse-index (read-only)

This index maps product IDs to the ingredient pairs that can create them.
It enables efficient lookup of "what ingredients make this product?" and
is essential for target-to-base DFS traversal.
*/
type pair struct{ a, b int } // Represents an ingredient pair (a,b)
type revIndex map[int][]pair // Maps product ID to all ingredient pairs

var revIdx revIndex // Global reverse index: productID → pairs

// BuildReverseIndex creates a reverse mapping from products to their ingredient pairs.
// This should be called once at startup before running DFS algorithms.
// The index is sorted by ingredient tier sum (lower tiers first) to optimize search.
func BuildReverseIndex(g IndexedGraph) {
    idx := make(revIndex)

    // First pass: collect all ingredient pairs for each product
    for a, nbrs := range g.Edges {
        for _, e := range nbrs {
            // Store pair in normalized order (smaller ID first)
            p := pair{a: min(a, e.PartnerID), b: max(a, e.PartnerID)}
            idx[e.ProductID] = append(idx[e.ProductID], p)
        }
    }

    // Second pass: sort pairs by total tier (complexity) of ingredients
    // This prioritizes simpler ingredients during search
    for _, list := range idx {
        sort.Slice(list, func(i, j int) bool {
            ti := getElementTier(g.IDToName[list[i].a]) +
                getElementTier(g.IDToName[list[i].b])
            tj := getElementTier(g.IDToName[list[j].a]) +
                getElementTier(g.IDToName[list[j].b])
            return ti < tj
        })
    }
    revIdx = idx // Set the global variable
}

/*
-----
Single-path DFS (recursive)

This algorithm finds a single path from a target element to base elements using
depth-first search with caching and pruning optimizations.

```

```

/*
var canReachBaseCache map[int]bool // Memoization cache: elementID → can reach base?

// findPathToBaseCnt is a recursive DFS function that finds a path from an element to base
elements.
// Parameters:
//   - id: Current element ID being processed
//   - depth: Current recursion depth
//   - maxDepth: Maximum recursion depth limit to prevent stack overflow
//   - g: The indexed graph containing all element relationships
//   - recipes: Output map to store the found recipe steps
//   - visit: Map to track visited elements (prevents cycles)
//   - counter: Pointer to count nodes visited (for statistics)
//
// Returns:
//   - bool: True if a path to base elements was found, false otherwise
func findPathToBaseCnt(
    id, depth, maxDepth int,
    g IndexedGraph,
    recipes ProductToIngredients,
    visit map[int]bool,
    counter *int,
) bool {
    // Stop if we've gone too deep (prevents stack overflow)
    if depth > maxDepth {
        return false
    }

    // Check the cache for previous results (memoization)
    if res, ok := canReachBaseCache[id]; ok {
        return res
    }

    // Detect cycles in the current path
    if visit[id] {
        canReachBaseCache[id] = false
        return false
    }

    // Mark as visited temporarily for this path
    visit[id] = true
    defer func() { visit[id] = false }()
    *counter++ // Count this node as visited

    // Check if current element is a base element (success case)
    name := g.IDToName[id]
    for _, b := range BaseElements {
        if name == b {
            canReachBaseCache[id] = true
            return true
        }
    }

    // Find all ingredient pairs that can make this element
}

```

```

ing := findIngredientsFor(id, g)

// Sort ingredients by total tier (simpler ingredients first)
sort.Slice(ing, func(i, j int) bool {
    ti := getElementTier(g.IDToName[ing[i][0]]) + getElementTier(g.IDToName[ing[i][1]])
    tj := getElementTier(g.IDToName[ing[j][0]]) + getElementTier(g.IDToName[ing[j][1]])
    return ti < tj
})

// Recursively try each ingredient pair
for _, pr := range ing {
    a, b := pr[0], pr[1]

    // Try to find paths from both ingredients to base elements
    if findPathToBaseCnt(a, depth+1, maxDepth, g, recipes, visit, counter) &&
        findPathToBaseCnt(b, depth+1, maxDepth, g, recipes, visit, counter) {

        // Record the successful recipe step
        recipes[name] = RecipeStep{
            Combo: IngredientCombo{
                A: g.IDToName[a],
                B: g.IDToName[b],
            },
        }
        canReachBaseCache[id] = true
        return true
    }
}

// No valid path found
canReachBaseCache[id] = false
return false
}

// DFSBuildTargetToBase performs a target-to-base DFS search to find a single valid recipe
path.
// It starts from the target element and works backward to find constituent ingredients
// until reaching base elements.
// Parameters:
//   - target: Name of the target element to find a recipe for
//   - g: The indexed graph containing all element relationships
//
// Returns:
//   - ProductToIngredients: Map of products to their ingredient recipes
//   - int: Count of nodes visited during the search
func DFSBuildTargetToBase(target string, g IndexedGraph) (ProductToIngredients, int) {
    targetID := g.NameToID[target]
    recipes := make(ProductToIngredients)
    visited := make(map[int]bool)

    // Initialize cache with base elements (they can reach themselves)
    canReachBaseCache = map[int]bool{}
    for _, b := range BaseElements {
        canReachBaseCache[g.NameToID[b]] = true
    }
}

```

```

    }

    // Start DFS with nodes counter
    nodes := 0

    // First try with reasonable depth limit
    if !findPathToBaseCnt(targetID, 0, 1000, g, recipes, visited, &nodes) {
        // If that fails, try again with much higher limit
        visited = map[int]bool{}
        findPathToBaseCnt(targetID, 0, 10000, g, recipes, visited, &nodes)
    }

    return recipes, nodes
}

/*
-----
Multi-path DFS (iterative/parallel)
*/
// RangeDFSPaths finds up to maxPaths unique DFS recipes, exploring each top-level
//
// For each reverse-combination (root pair) that produces the target, a separate
// goroutine is launched to recursively explore all possible ingredient paths back
// to base elements. A bounded worker pool (limited to NumCPU()) ensures controlled
// parallelism.
//
// Each path is deduplicated using a hash signature to guarantee uniqueness.
// Once maxPaths unique results are found, all active searches are cancelled early.
func RangeDFSPaths(target string, maxPaths int, g IndexedGraph) ([]RecipeStep, int) {
    targetID := g.NameToID[target]
    roots := revIdx[targetID]

    var (
        out      []RecipeStep
        seenSig = make(map[uint64]struct{})
        mu      sync.Mutex
        nodes   int64
    )

    // bound concurrent workers
    sem := make(chan struct{}, runtime.NumCPU())
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    var wg sync.WaitGroup

    // recursive DFS
    var dfs func(id int, path [][]int, visited map[int]bool)
    dfs = func(id int, path [][]int, visited map[int]bool) {
        select {
        case <-ctx.Done():
            return
        default:
        }
    }
}

```

```

        atomic.AddInt64(&nodes, 1)

        if isBaseID(id, g) {
            sig := hashPath(path)
            mu.Lock()
            if len(out) < maxPaths {
                if _, dup := seenSig[sig]; !dup {
                    seenSig[sig] = struct{}{}
                    out = append(out, buildRecipeStepFromPath(path, targetID, g))
                    if len(out) == maxPaths {
                        cancel()
                    }
                }
            }
            mu.Unlock()
            return
        }

        if visited[id] {
            return
        }
        visited[id] = true
        defer func() { visited[id] = false }()

        for _, pr := range revIdx[id] {
            newPath := append(path, []int{pr.a, pr.b, id})
            dfs(pr.a, newPath, visited)
            dfs(pr.b, newPath, visited)
        }
    }

    // spawn one worker per root pair
    for _, pr := range roots {
        sem <- struct{}{} // acquire slot
        wg.Add(1)          // one Add per goroutine
        go func(pr pair) {
            defer wg.Done()
            defer func() { <-sem }()
            // release slot

            visited := make(map[int]bool)
            initial := [][]int{{pr.a, pr.b, targetID}}
            dfs(pr.a, initial, visited)
            dfs(pr.b, initial, visited)
        }(pr)
    }

    wg.Wait()
    return out, int	atomic.LoadInt64(&nodes))
}

// isBaseID returns true if id corresponds to one of the base elements.
func isBaseID(id int, g IndexedGraph) bool {
    for _, b := range BaseElements {

```

```

        if id == g.NameToID[b] {
            return true
        }
    }
    return false
}

```

c. [helper.go](#)

```

package recipeFinder

import (
    "container/list"
    "hash/fnv"
)

/*
-----
Helper & util (GENERAL)
*/
// hashPath generates a hash signature for a specific recipe path.
// This is used to deduplicate paths that are functionally equivalent.
func hashPath(p [][]int) uint64 {
    h := fnv.New64a()
    var buf [4]byte

    put := func(v int) {
        buf[0] = byte(v)
        buf[1] = byte(v >> 8)
        buf[2] = byte(v >> 16)
        buf[3] = byte(v >> 24)
        _'_' = h.Write(buf[:])
    }

    for _, t := range p {
        a, b := min(t[0], t[1]), max(t[0], t[1])
        put(a)
        put(b)
        put(t[2])
    }

    return h.Sum64()
}

func buildRecipeStepFromPath(path [][]int, targetID int, g IndexedGraph) RecipeStep {
    if len(path) == 0 {
        return RecipeStep{}
    }
    strPath := make([][]string, len(path))
    for i, t := range path {
        strPath[i] = []string{
            g.IDToName[t[0]],

```

```

        g.IDToName[t[1]],
        g.IDToName[t[2]],
    }
}
last := path[len(path)-1]
return RecipeStep{
    Combo: IngredientCombo{
        A: g.IDToName[last[0]],
        B: g.IDToName[last[1]],
    },
    Path: strPath,
}
}

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

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

/*
-----
Helper & util (BFS)
*/
// Helper to extract a path from single-recipe BFS result
func extractPathFromRecipes(targetID int, recipes ProductToIngredients, g IndexedGraph)
RecipeStep {
    // Build a path by following the recipe chain from target to base elements
    var path [][]string
    current := g.IDToName[targetID]

    // Track visited elements to avoid cycles
    visited := make(map[string]bool)

    // Walk the recipe chain
    for {
        if visited[current] {
            break // Avoid cycles
        }
        visited[current] = true

        recipe, exists := recipes[current]
        if !exists {
            break
        }

        a := recipe.Combo.A

```

```

b := recipe.Combo.B

// Add this step to the path
path = append(path, []string{a, b, current})

// If both ingredients are base elements, we're done
aIsBase := isBaseElement(a)
bIsBase := isBaseElement(b)

if aIsBase && bIsBase {
    break
}

// Continue with a non-base ingredient
if !aIsBase {
    current = a
} else {
    current = b
}
}

// Reverse the path since we built it backwards
for i, j := 0, len(path)-1; i < j; i, j = i+1, j-1 {
    path[i], path[j] = path[j], path[i]
}

// Convert to RecipeStep format
return RecipeStep{
    Combo: IngredientCombo{
        A: recipes[g.IDToName[targetID]].Combo.A,
        B: recipes[g.IDToName[targetID]].Combo.B,
    },
    Path: path,
}
}

// Helper function to convert map keys to a slice of ints
func mapKeysToSlice(m map[int]bool) []int {
    result := make([]int, 0, len(m))
    for k := range m {
        result = append(result, k)
    }
    return result
}

// Helper function to convert map keys to element names
func mapKeysToNameSlice(m map[int]bool, g IndexedGraph) []string {
    result := make([]string, 0, len(m))
    for k := range m {
        result = append(result, g.IDToName[k])
    }
    return result
}

```

```

// Helper function to convert queue to a slice of ints
func queueToSlice(q *list.List) []int {
    result := make([]int, 0, q.Len())
    for e := q.Front(); e != nil; e = e.Next() {
        result = append(result, e.Value.(int))
    }
    return result
}

// Helper function to convert queue to a slice of names
func queueToStringSlice(q *list.List, g IndexedGraph) []string {
    result := make([]string, 0, q.Len())
    for e := q.Front(); e != nil; e = e.Next() {
        id := e.Value.(int)
        result = append(result, g.IDToName[id])
    }
    return result
}

// copyMap creates a deep copy of a map of product IDs to parent/partner IDs
func copyMap(m map[int]struct{ ParentID, PartnerID int }) map[int]struct{ ParentID,
PartnerID int } {
    result := make(map[int]struct{ ParentID, PartnerID int }, len(m))
    for k, v := range m {
        result[k] = v // struct is copied by value
    }
    return result
}

// prevIDsToNames converts the integer IDs in the prevIDs map to their string names
func prevIDsToNames(m map[int]struct{ ParentID, PartnerID int }, g IndexedGraph)
map[string]struct{ A, B string } {
    result := make(map[string]struct{ A, B string }, len(m))
    for productID, info := range m {
        productName := g.IDToName[productID]
        result[productName] = struct{ A, B string }{
            A: g.IDToName[info.ParentID],
            B: g.IDToName[info.PartnerID],
        }
    }
    return result
}

/*
-----
Helper & util (DFS)
*/
// findIngredientsFor returns all ingredient pairs that can create a specific product.

func findIngredientsFor(productID int, g IndexedGraph) [][]int {
    var res [][]int

    // Scan through the entire graph looking for combinations that produce our target
    for aID, nbrs := range g.Edges {

```

```

        for _, e := range nbrs {
            if e.ProductID == productID {
                // Found a combination that produces our target
                res = append(res, []int{aID, e.PartnerID})
            }
        }
    }

    return res
}

```

### 4.3. Tata Cara Penggunaan Program

#### a. Interface Website

The screenshot shows the homepage of the "Little Alchemy 2 Recipe Finder". The title "Little Alchemy 2 Recipe Finder" is at the top center. On the left is a logo consisting of two squares. On the right is a "Scrape All" button. Below the title are three tabs under "Choose Algorithm": "Breadth-First Search (BFS)" (selected), "Depth-First Search (DFS)", and "Bidirectional Search". Below that are two tabs under "Recipe Search Mode": "Shortest Recipe" (selected) and "Multiple Recipes". A "Target Element" input field contains the placeholder "Masukkan nama elemen". At the bottom is a yellow "Find Recipe(s)" button.

## Little Alchemy 2 Elements

[Go to Recipe Finder](#)

Search for an element...

### Starting

Element	Recipes
Air Starting	🔥 Fire (0) + 🌬 Mist (1)
Earth Starting	
Fire Starting	🔥 Fire (0) + 🍹 Alcohol (11) 🔥 Fire (0) + 🐫 Coal (9)
Water Starting	⚡ Heat (2) + ❄ Ice (9) ⚡ Heat (2) + ❄ Snow (7)

Search Time : 2.76 ms

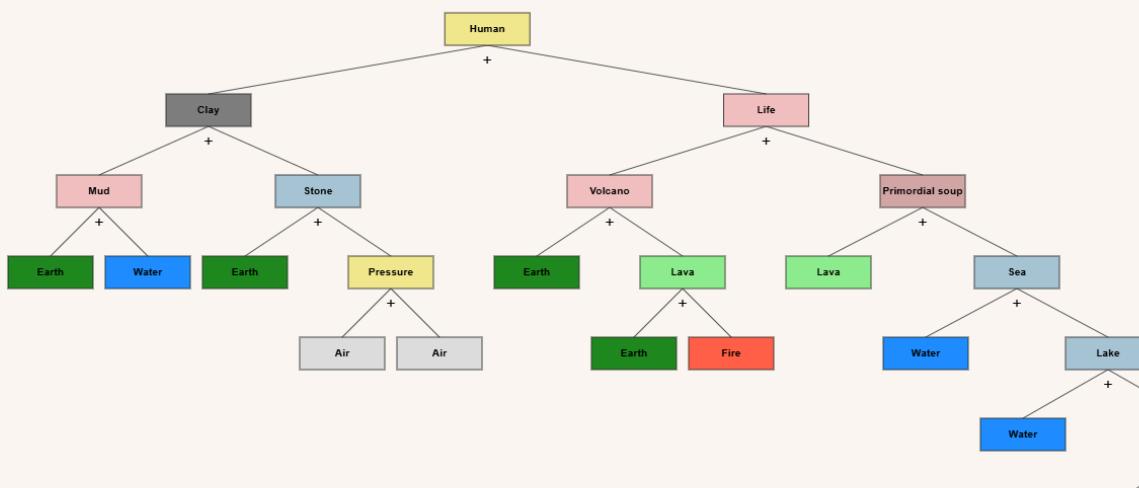
Visited Node: 80

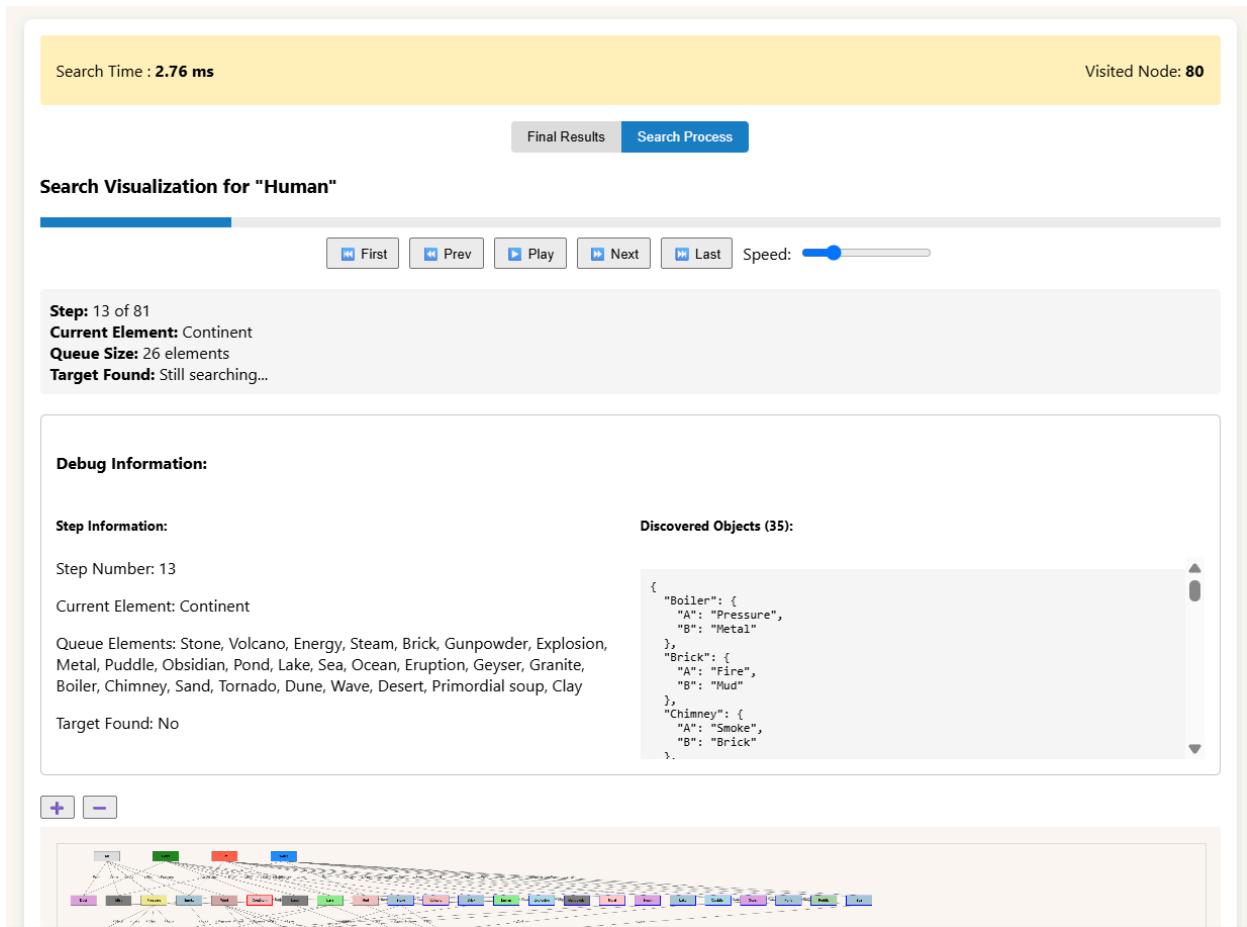
[Final Results](#) [Search Process](#)

### Found 1 recipe for Human

#### Recipe 1

+ -





## b. Fitur-Fitur yang Disediakan

### 1. Choose Algorithm

User dapat memilih algoritma apa yang ingin digunakan untuk mencari resep dari sebuah elemen. Algoritma yang tersedia pada website ini adalah BFS dan DFS

### 2. Recipe Search Mode

User dapat memilih ingin mencari resep sebuah elemen dengan jalur tercepat (shortest recipe) atau beberapa jalur (multiple recipe)

### 3. Target Element

User dapat memasukkan nama elemen (huruf depan harus huruf kapital) yang ingin dicari resepnya

#### 4. Little Alchemy 2 Catalog

User dapat melihat dan mencari daftar resep seluruh elemen yang merupakan hasil *scraping* dari website Little Alchemy 2

#### 5. Scrape All

User dapat melakukan *scraping* data tanpa harus menjalankan *source code* dari terminal

#### 6. Final Results

User dapat melihat hasil akhir elemen yang dicari dalam bentuk recipe tree

#### 7. Search Process

User dapat melihat process pembentukan recipe tree secara bertahap, Fitur ini hanya bisa dipakai saat single recipe pada algoritma BFS

### c. Cara Penggunaan Program

Untuk prerequisite keberjalanannya website, bisa dilihat pada README github kelompok [Semoga Gak Masuk UGD](#). Setelah melakukan prerequisite tersebut, program dapat dijalankan dengan mengikuti tata cara penggunaan sebagai berikut:

#### 1. Memilih algoritma

User diminta memilih algoritma melalui fitur Choose Algorithm, misalnya user memilih algoritma BFS

#### 2. Memilih metode pencarian

User memilih metode pencarian pada fitur Recipe Search Mode, misalnya user memilih multiple recipes. Kemudian user akan memasukkan berapa maksimal resep yang ingin ia cari

#### 3. Masukan nama elemen

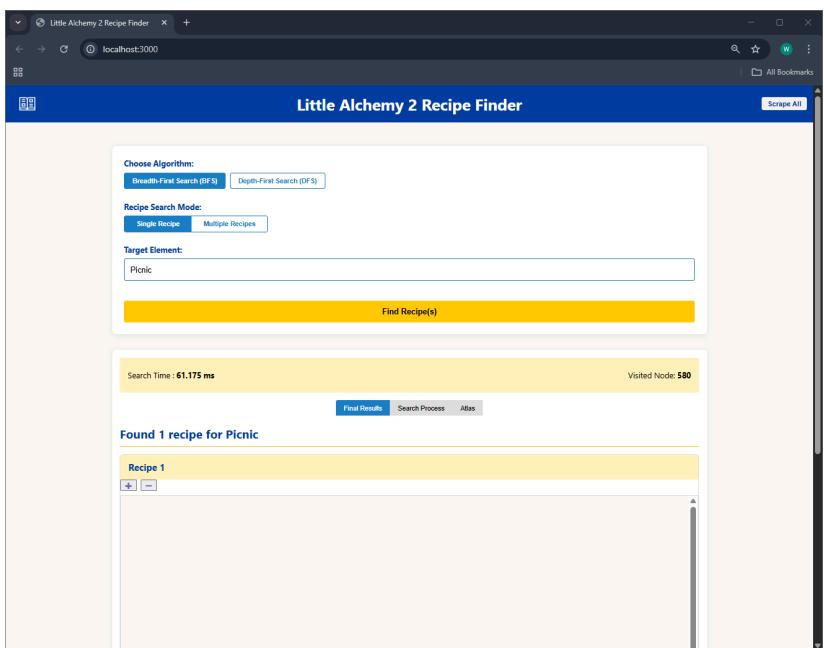
User akan memasukan nama elemen yang ingin dicari resepnya melalui input box.

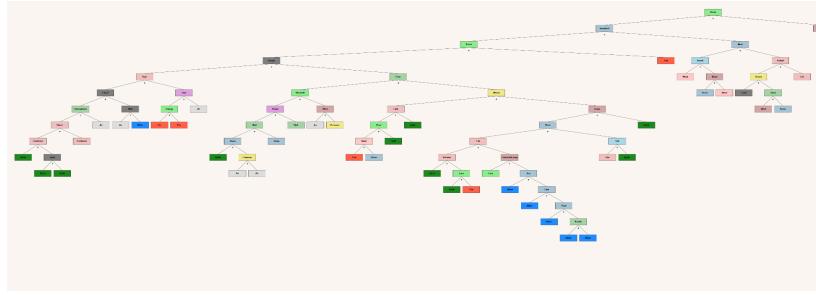
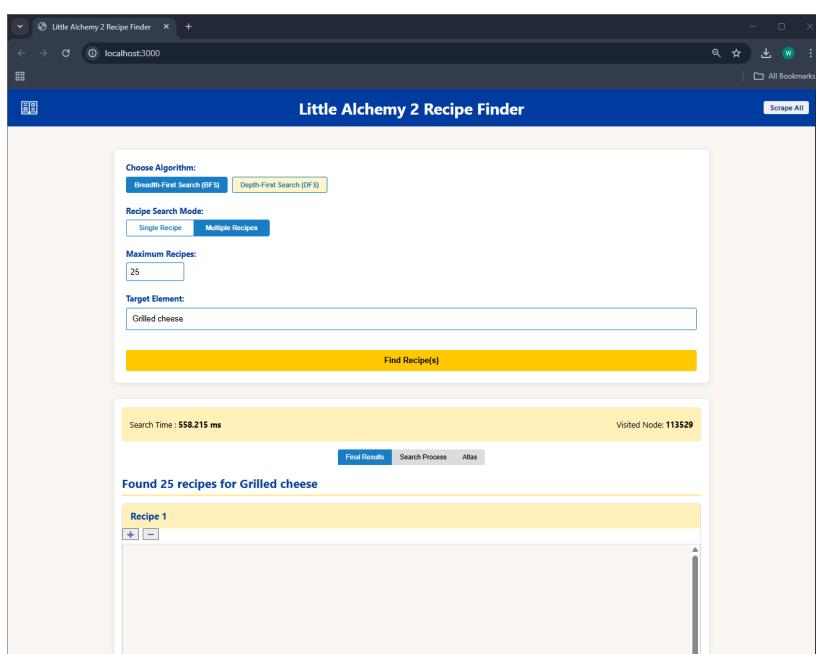
#### 4. Menekan tombol Find Recipe(s)

Setelah user mengisi input, user harus menekan tombol Find Recipe(s) dan menunggu hasil berupa tree

#### 4.4. Hasil Pengujian

**minimal 3** buah elemen dalam bentuk *screenshot* antarmuka. Variasikan setiap kasus dengan berbagai fitur yang diimplementasikan (Algoritma serta banyak *recipe* yang dicari).

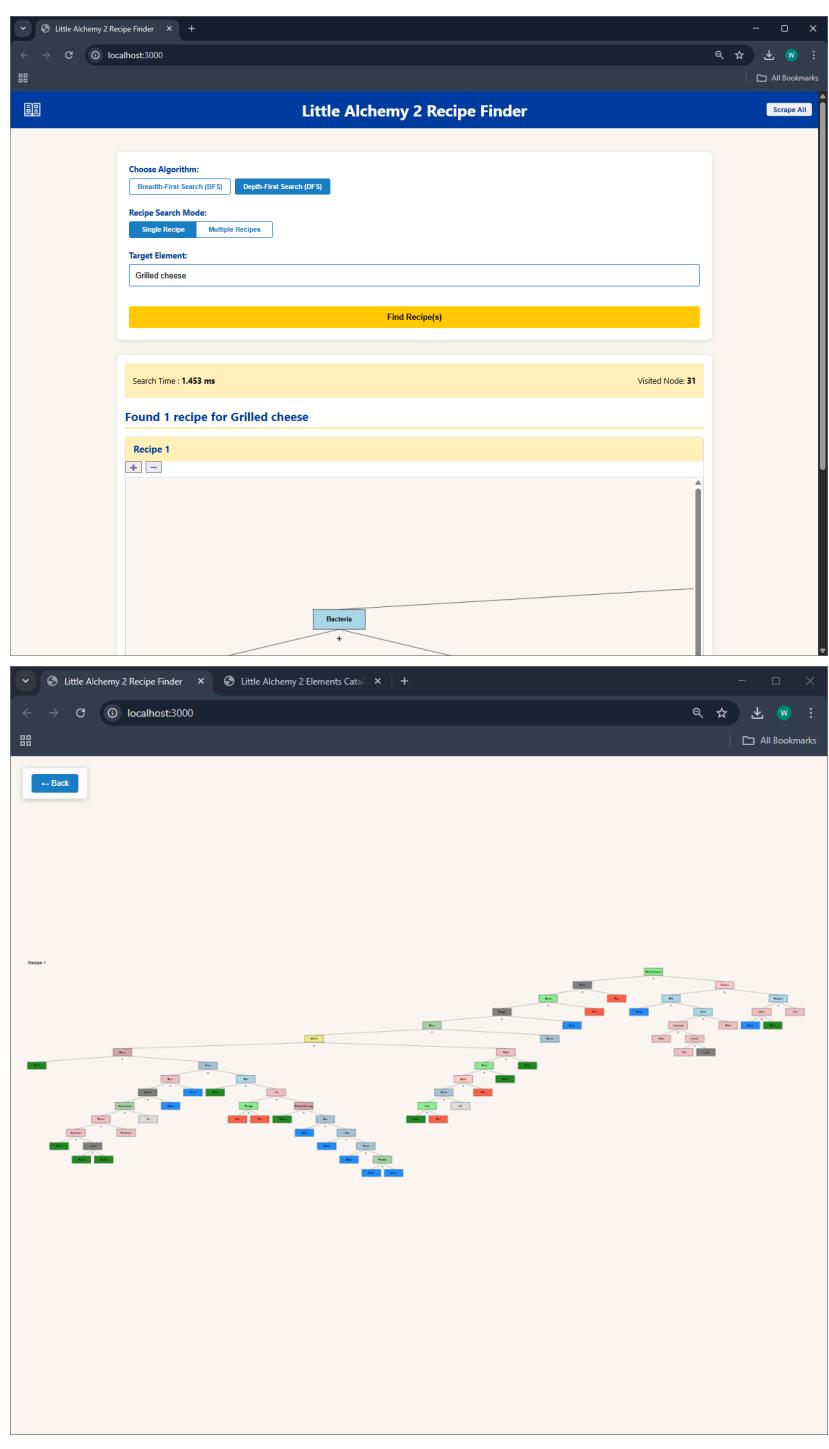
1	Metode: Single Recipe BFS  Target: Picnic	
---	---	---

		
2	<p>Metode: Multiple Recipe BFS</p> <p>Target: 25 Picnic</p>	 <p>The screenshot shows the 'Little Alchemy 2 Recipe Finder' application. It has a search interface with fields for 'Choose Algorithm' (Breadth-First Search (BFS) is selected), 'Recipe Search Mode' (Multiple Recipes is selected), 'Maximum Recipes' (set to 25), and 'Target Element' (set to 'Grilled cheese'). A yellow button labeled 'Find Recipe(s)' is present. Below the interface, a status bar indicates 'Search Time : 558.215 ms' and 'Visited Node: 113529'. A section titled 'Found 25 recipes for Grilled cheese' displays a list of recipes, with 'Recipe 1' currently selected. The bottom part of the screenshot shows a grid of 12 small diagrams, each representing a different recipe path from basic elements to 'Grilled cheese'.</p>

3

Metode: Single Recipe DFS

Target: Grilled cheese



4

Metode: Multiple Recipe DFS

Target: Grilled cheese

The screenshot shows two browser windows. The top window is titled "Little Alchemy 2 Recipe Finder" and displays a search interface. It has sections for "Choose Algorithm" (Breadth-First Search (BFS) is selected), "Recipe Search Mode" (Multiple Recipes is selected), "Maximum Recipes" (set to 25), and a "Target Element" input field containing "Grilled cheese". A yellow "Find Recipe(s)" button is at the bottom. Below this, a message says "Search Time: 2830.886 ms" and "Visited Node: 9030". It then lists "Found 25 recipes for Grilled cheese", starting with "Recipe 1". The bottom window is titled "Little Alchemy 2 Elements Catalog" and shows a grid of various elements represented by small icons.

## **4.5. Analisis Pengujian**

### **1. Metode: Single Recipe BF**

Target: Picnic

Metode Single Recipe BFS berhasil menemukan jalur pembuatan resep untuk Picnic dengan menggunakan hanya satu kombinasi resep untuk setiap produk turunan. Pencarian dilakukan secara mendatar (breadth-first), memastikan bahwa jalur yang ditemukan merupakan salah satu jalur terpendek dari bahan dasar ke target. Hasilnya menunjukkan bahwa metode ini mampu menyusun langkah-langkah pembuatan Picnic secara efisien dan deterministik, namun tidak mengeksplorasi kemungkinan jalur alternatif.

### **2. Metode: Multiple Recipe BF**

Target: 25 Picnic

Metode Multiple Recipe BFS juga berhasil menyusun resep untuk 25 Picnic, namun dengan pendekatan yang berbeda. Karena menggunakan multiple recipe, metode ini tidak hanya mempertimbangkan satu jalur pembuatan, melainkan mengeksplorasi berbagai alternatif jalur dari bahan dasar hingga produk akhir. Hasilnya menunjukkan bahwa metode ini mampu menghasilkan banyak variasi resep (multi-path) untuk mencapai target, selama masih dalam batas maksimum jumlah jalur yang ditentukan (dalam hal ini, 25). Metode BFS menjamin bahwa jalur-jalur yang ditemukan adalah yang paling pendek secara tingkat kedalaman sebelum berlanjut ke eksplorasi lebih dalam.

### **3. Metode: Single Recipe DFS**

Target: Grilled cheese

Pada metode ini, pencarian dilakukan secara mendalam (depth-first), namun tetap hanya mempertimbangkan satu kombinasi resep untuk setiap produk. Hasilnya menunjukkan bahwa metode ini berhasil membentuk resep lengkap untuk membuat Grilled cheese, dengan kecenderungan eksplorasi yang lebih cepat menuju jalur tertentu. Metode ini cocok untuk menemukan solusi dengan kedalaman maksimal lebih dahulu, meskipun tidak menjamin bahwa jalur yang ditemukan adalah yang paling pendek.

### **4. Metode: Multiple Recipe DFS**

Target: Grilled cheese

Metode ini menunjukkan keberhasilan dalam menyusun berbagai jalur pembuatan Grilled cheese dengan memanfaatkan seluruh alternatif resep yang tersedia. Dengan pendekatan DFS, pencarian mendahuluikan jalur yang lebih dalam sebelum berpindah ke cabang lain, sehingga mampu menjelajahi kombinasi-kombinasi unik yang mungkin tidak ditemukan oleh BFS. Metode ini efektif untuk menghasilkan variasi resep yang lebih kreatif, meskipun bisa jadi memerlukan kontrol batas eksplorasi untuk mencegah terlalu dalamnya pencarian pada satu cabang.

Berdasarkan hasil pengujian yang telah dilakukan, algoritma DFS menunjukkan performa yang lebih efisien dibandingkan dengan BFS dalam konteks pencarian resep pada proyek ini. Hal ini dapat dilihat dari dua metrik utama, yaitu jumlah simpul yang dilalui dan waktu eksekusi.

Terkait kompleksitas algoritma:

- BFS dalam implementasi ini memiliki kompleksitas waktu mendekati  $O(b^d)$ , di mana b adalah branching factor dan d adalah kedalaman solusi. Karena BFS mengeksplorasi semua jalur level demi level, kompleksitas ruang juga menjadi  $O(b^d)$  karena banyaknya node yang disimpan pada setiap level eksplorasi.
- DFS memiliki kompleksitas waktu  $O(b^d)$  juga dalam kasus terburuk, namun dengan optimisasi melalui pemotongan jalur duplikat, early termination, dan pendekatan iteratif berbasis stack, DFS secara praktis menunjukkan efisiensi yang lebih tinggi dalam kasus pencarian terbatas (`maxPaths` kecil). Kompleksitas ruang DFS cenderung lebih rendah, yakni  $O(d)$ , karena hanya perlu menyimpan path saat ini.

Kami menemukan jumlah simpul yang dilalui pada BFS jauh lebih banyak dibandingkan DFS. Hal ini konsisten pada hampir semua skenario. Ini juga dipengaruhi oleh perbedaan pendekatan, yaitu BFS dengan *bottom-up approach*, sedangkan DFS dengan *top-down approach*.

Secara keseluruhan, meskipun BFS menyediakan pencarian yang lebih sistematis dan cocok untuk pencarian jalur terpendek, pada kasus pencarian multi-path terbatas seperti proyek ini, DFS terbukti lebih optimal dan lebih hemat sumber daya.

## **BAB V – KESIMPULAN, SARAN, DAN REFLEKSI TUGAS BESAR 2**

### **5.1. Kesimpulan**

Dalam pembangunan “Little Alchemy 2 Recipe Finder” berbasis web ini, telah berhasil diimplementasikan dengan algoritma Breadth First Search (BFS) dan Depth First Search (DFS) untuk menemukan rute terpendek dari recipe sebuah elemen. Dengan menggunakan teknik *scraping* dan analisis graf, website ini mampu memberikan solusi dalam waktu yang relatif cepat. Selain itu, antarmuka pengguna yang responsif dapat meningkatkan pengalaman pengguna secara keseluruhan.

### **5.2. Saran**

Saran untuk pengembangan sebuah website adalah melakukan optimasi secara terus-menerus guna menjaga kinerja dan responsivitas aplikasi, baik dari segi algoritma pencarian maupun penggunaan penyimpanan. Selain itu, disarankan untuk terus memperdalam pengetahuan tentang teknik *scraping* dan konkurensi karena keduanya memiliki peran penting dalam pengembangan website. Selalu meningkatkan fitur website untuk meningkatkan kualitas dan kepuasan pengguna secara berkelanjutan.

### **5.3. Refleksi Tugas Besar 2**

Proses pembuatan website Little Alchemy 2 Recipe Finder ini telah memberikan pengalaman berharga tentang *scraping* data, algoritma BFS dan DFS, serta pengelolaan konkurensi dalam lingkungan pemrograman. Melalui tugas besar 2 ini, telah dipelajari bagaimana mengintegrasikan berbagai teknologi dan algoritma untuk mencapai solusi yang efektif. Tugas besar 2 ini juga menjadi *reminder* bahwa pembelajaran kontinu dan kerja keras adalah kunci untuk meraih kesuksesan dalam pengembangan perangkat lunak.

## DAFTAR PUSTAKA

Institut Teknologi Bandung. (2024). *Spesifikasi Tugas Besar 2 Stima 2024/2025* [Google Document]. Diambil dari

<https://docs.google.com/document/d/1aQB5USxfUCBfHmYjKl2wV5WdMBzDEyojE5y xvBO3pvc/edit?tab=t.0>

Munir, R. (2025). Strategi Algoritma 2024/2025. Diambil dari <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/stima24-25.htm>

React Team. (n.d.). *React – A JavaScript library for building user interfaces*. Diambil dari <https://react.dev/>

The Go Authors. (n.d.). *The Go Programming Language Documentation*. Diambil dari <https://go.dev/doc/>

## LAMPIRAN

### A. Pranala Repository:

[https://github.com/wiwekaputera/Tubes2\\_SemogaGaMasukUGD](https://github.com/wiwekaputera/Tubes2_SemogaGaMasukUGD)

### B. Pranala Video:

<https://youtu.be/NXIheyWoVeU?si=FIIFFoXahMgGl0hP>

### C. Tabel Ketercapaian

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.	✓	