LAPORAN TUGAS BESAR 2
IF2211 STRATEGI ALGORITMA

Pemanfaatan Algoritma *BFS* dan *DFS* dalam Pencarian Recipe pada Permainan

Little Alchemy 2

Disusun Oleh:

M. Rayhan Farrukh	13523035
Azfa Radhiyya Hakim	13523115
Ferdin Arsenarendra Purtadi	13523117

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2025

DAFTAR ISI

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.

II. Landasan Teori

2.1. Path Traversal

Path Traversal atau *Graph Traversal* secara umum adalah proses menemukan dan mengikuti suatu rute dari satu titik awal ke titik tujuan dalam sebuah struktur data, seperti graf atau pohon. Ini merupakan konsep fundamental dalam computer science yang bertujuan untuk mengidentifikasi apakah sebuah jalur ada, dan menemukan jalur berdasarkan kriteria tertentu (misalnya jalur terpendek). Aplikasi path traversal sangat luas, yang mencakup sistem navigasi, route planning dalam topologi jaringan, hingga pemecahan masalah artificial intelligence.

Algoritma seperti Depth-First Search (DFS) dan Breadth-First Search (BFS) adalah contoh klasik dari algoritma graph traversal. Algoritma-algoritma seperti DFS, BFS, dan Bidirectional Search adalah algoritma yang akan dibahas untuk selanjutnya untuk melakukan penelusuran jalur dalam menentukan *recipe* sebuah elemen pada permainan *Little Alchemy 2* secara efisien dan efektif.

2.2. BFS

Breadth-First Search (BFS) adalah salah satu algoritma *graph traversal* yang fundamental. Algoritma ini menjelajahi graf secara sistematis dengan mengunjungi semua simpul tetangga pada kedalaman atau level yang sama sebelum melanjutkan ke simpul-simpul pada kedalaman berikutnya.

BFS menjelajahi graf dimulai dari sebuah simpul awal (sumber) dan mengunjungi semua *direct neighbour (*tetangga langsung*)* dari simpul tersebut. Kemudian, untuk setiap tetangga tersebut, BFS kemudian mengunjungi tetangga-tetangga langsung dari mereka yang belum pernah dikunjungi sebelumnya, dan begitu seterusnya. Untuk melacak simpul-simpul yang akan dikunjungi, BFS menggunakan struktur data *queue*, yaitu dengan prinsip *First-In, First-Out,* dimana simpul yang dijelajahi adalah simpul pertama yang ditemukan.

Algoritma ini menjamin penemuan jalur terpendek dari simpul awal ke semua simpul lain dalam graf tanpa bobot (*unweighted graph*). BFS juga *complete*, artinya ia akan menemukan solusi jika solusi tersebut ada.

2.3. DFS

Depth-First Search (DFS) adalah algoritma *graph traversal* yang melakukan penjelajahan sejauh mungkin pada satu cabang sebelum melakukan *backtracking* untuk berpindah ke cabang lainnya. Berbeda dengan BFS yang menjelajah *layer* demi *layer*, DFS akan memilih satu jalur dan mengikutinya hingga mencapai simpul terakhir atau simpul yang sudah pernah dikunjungi, baru kemudian kembali ke simpul sebelumnya untuk menjelajahi jalur lain yang belum dieksplorasi.

Proses DFS dimulai dari simpul awal, kemudian bergerak ke salah satu tetangganya yang belum dikunjungi. Dari tetangga tersebut, proses yang sama diulang kembali (rekursif). Jika mencapai simpul

yang semua tetangganya sudah dikunjungi atau simpul yang tidak memiliki tetangga, DFS akan melakukan backtrack ke simpul sebelumnya dan mencoba menjelajahi cabang lain. Untuk mengelola urutan kunjungan dan proses *backtracking*, DFS menggunakan struktur data *stack*, dengan prinsip *Last-in First-out*, biasanya melalui pemanggilan rekursif.

DFS sangat berguna untuk berbagai aplikasi, contohnya untuk mendeteksi siklus dalam graf, melakukan *topological sorting*, dan memecahkan masalah yang dapat direpresentasikan sebagai pencarian mendalam, contohnya dalam penyelesaian labirin.

2.4. Bidirectional

Bidirectional Search adalah algoritma *graph traversal* yang bekerja dengan cara menjalankan dua pencarian secara sekaligus, satu dimulai dari *start node* menuju *goal node*, dan yang satu lagi dimulai dari simpul tujuan menuju simpul awal. Pencarian ini dianggap selesai ketika kedua proses pencarian bertemu di satu simpul.

Ide utama di balik *bidirectional search* adalah mengurangi kompleksitas pencarian secara signifikan. Jika kita mengasumsikan *branching factor* dari graf adalah b dan jarak dari awal ke tujuan adalah d, maka pencarian satu arah seperti BFS atau DFS mungkin perlu menjelajahi sekitar b^d simpul. Dengan pencarian dua arah, masing-masing pencarian idealnya hanya perlu menjelajah hingga kedalaman $d/2$. Oleh karena itu, total simpul yang dieksplorasi dapat berkurang menjadi $2 * b^{d/2}2$, yang jauh lebih kecil daripada b^d, terutama untuk nilai d yang besar.

2.5. Website

Website adalah kumpulan halaman web yang saling terhubung dan berada di bawah satu domain, yang dapat diakses melalui internet menggunakan browser. Website digunakan untuk memvisualisasi hasil *tree* yang diperoleh melalui algoritma dan metode yang dipakai. Arsitektur aplikasi yang kami gunakan dibagi menjadi dua bagian, yaitu sisi frontend dan sisi backend.

1. Frontend

   Frontend dibangung menggunakan Next.js dengan bantuan Typescript untuk membuat interface yang dinamis, responsif, dan dapat dikembangkan secara terstruktur. Framework ini memanfaatkan *server side* dan optimasi build yang mendukung performa.

2. Backend

   Backend dibangun menggunakan bahasa pemrograman Golang. Bahasa ini dipilih karena memiliki performa yang tinggi, efisiensi dalam penggunaan memori, serta dukungan bawaan untuk concurrency (proses paralel) dengan *multithread* yang sangat baik. Hal ini membuat Golang sangat cocok untuk membangun sistem backend yang membutuhkan komunikasi real-time seperti WebSocket, karena mampu memproses banyak permintaan secara bersamaan

dengan cepat dan stabil. Backend bertugas mengelola data elemen, menjalankan algoritma pencarian, dan menyediakan API untuk frontend. Frontend di hosting melalui layanan Vercel, yang memiliki dukungan native terhadap framework Next.js serta fitur manajemen domain dan deployment otomatis langsung dari repository Git. Sementara itu, backend hosting menggunakan platform Railway, yang mendukung container Docker, websocket, dan mampu menjalankan API berbasis Golang secara efisien.

2.6. Docker

Docker adalah platform perangkat lunak sumber terbuka yang digunakan untuk mempermudah proses pengembangan, penyebaran, dan menjalankan aplikasi di dalam kontainer. Container Docker adalah unit mandiri yang berisi semua yang dependencies yang diperlukan agar sebuah aplikasi dapat berjalan secara konsisten di berbagai environment. Untuk menjaga kelancaran dan konsistensi dalam proses pengembangan maupun deployment di berbagai lingkungan, aplikasi yang dibuat dikemas menggunakan Docker. Dengan penggunaan Docker, baik bagian frontend maupun backend dapat dijalankan dalam container yang terisolasi, sehingga memudahkan setup dan pengujian baik secara lokal maupun saat akan di *deploy*.

2.7. Websocket

WebSocket adalah protokol komunikasi jaringan yang menyediakan saluran komunikasi full-duplex dan interaktif antara server dan client melalui koneksi TCP yang stabil. WebSocket memungkinkan komunikasi dua arah secara real-time dengan latensi rendah. Protokol ini dirancang untuk mengatasi keterbatasan dari protokol HTTP tradisional yang bersifat satu arah dan tidak efisien untuk komunikasi real-time.

III. Analisis Pemecahan Masalah

3.1. Backend

3.1.1. Data Scraper

Untuk mengumpulkan data yang diperlukan oleh perangkat lunak, diperlukan sistem yang mampu mengekstrak informasi dari sumber eksternal secara otomatis, yaitu dengan scraping menggunakan go. Skrip yang kami buat melakukan proses berdasarkan beberapa tahap sebagai berikut.

1. Mendapatkan Daftar Elemen:
   Skrip pertama-tama mengakses halaman daftar elemen permainan di situs Little Alchemy 2. Elemen-elemen ini diambil dari berbagai selector dalam HTML (div, ul, li, etc) halaman tersebut. Elemen-elemen dasar yang harus ada dalam permainan ditambahkan ke dalam daftar elemen jika belum ada. Elemen-elemen ini kemudian dipastikan untuk tetap unik, dan tidak ada elemen yang duplikat.

2. Menyaring dan Menormalkan Resep:
   Setiap elemen kemudian diproses secara paralel menggunakan goroutine untuk mempercepat pengumpulan resep sekaligus membersihkan nama element.

3. Mengeksekusi Scraping secara Paralel:
   Skrip mengumpulkan resep-resep untuk setiap elemen secara paralel, dengan batasan pada jumlah goroutine yang aktif sekaligus untuk mencegah overloading server. Proses ini dilakukan dengan menggunakan channel dan sync.

4. Menghitung Tier Elemen:
   Setelah semua resep dikumpulkan, skrip ini menghitung tingkat tier dari masing-masing elemen

5. Menyimpan Data ke dalam Format JSON:
   Hasil akhirnya adalah sebuah file JSON yang berisi data elemen beserta resep-resep dan tier mereka.

3.1.2. Pemetaan Masalah

Berdasarkan hasil *scraping*, data-data elemen dan resep yang didapat kemudian disimpan di dalam program sebagai sebuah *map* dengan *key* berupa string dan *value* berupa sebuah struct yang menyimpan data-data elemen—yaitu nama, resep dan *tier* nya—dimana resep disimpan sebagai matriks n x 2, dengan n adalah jumlah resep berbeda dan 2 merepresentasikan jumlah elemen pada satu resep. Masing-masing elemen pada matriks resep disimpan sebagai *string* nama, sehingga dapat digunakan sebagai representasi graf . Struktur data *map* digunakan agar *lookup* resep ketika melakukan *traversal* pada graf memiliki kompleksitas O(1).

Untuk mencari solusi, masing-masing algoritma akan melakukan *traversal* yang dimulai dari elemen target sebagai *starting node*, dan menjadikan elemen dasar sebagai *goal node* untuk masing-masing percabangan, artinya penelusuran masing-masing cabang akan berhenti jika pada cabang tersebut ditemukan elemen dasar. Kemudian, untuk menyimpan solusi yang valid dari penelusuran, masing-masing algoritma akan menyimpan elemen-elemen yang dibutuhkan pada resep ke dalam sebuah struktur data *tree* yang merepresentasikan sebuah resep. Untuk kasus *multiple recipe*, masing-masing *tree* akan disimpan kedalam sebuah senarai yang merupakan *return type* utama dari algoritma.

3.1.3. BFS

Pencarian resep elemen dengan menggunakan BFS dilakukan dengan antrian (queue) secara iteratif dan berjalan secara multi-threaded (paralel) untuk mempercepat pencarian. Langkah-langkahnya adalah sebagai berikut:

- Persiapan data yang diperlukan, yaitu nama elemen yang ingin dicari (target), jumlah maksimal resep yang diminta.
- Periksa validitas elemen target. Jika elemen tidak ditemukan di map, atau ditemukan tapi tidak memiliki resep, kembalikan hasil kosong. Jika elemen adalah elemen dasar (basic element), langsung buat maka kembalikan pohon yang berisi satu elemen tersebut.
- Inisialisasi *search queue* dengan semua kombinasi resep awal untuk elemen target. Queue berisi path langkah pembuatan, elemen yang masih perlu dibuat (open), dan *depth* saat ini.
- Memulai sekumpulan *worker* (*thread*) untuk mengeksekusi pencarian. Setiap *worker* mengambil batch item dari *queue.*
- Jika rute sudah lengkap, maka *tree* dibangun, di-deduplikasi, dan disimpan.
- Jika jalur belum lengkap, maka satu elemen dari *open* di-*expand* menggunakan semua resepnya dan dimasukkan kembali ke *queue*. Selama proses berjalan, *queue* akan terus berkembang. Semua jalur pada level yang sama akan dieksplor sebelum masuk ke level berikutnya.
- Jika jumlah solusi sudah cukup atau *queue* habis, proses dihentikan.
- Semua hasil *recipe tree* yang valid dikembalikan sebagai hasil pencarian.

3.1.4. DFS

Pencarian resep elemen dengan menggunakan DFS dilakukan dengan menggunakan *stack* dengan menggunakan rekursi. Untuk langkah-langkahnya adalah sebagai berikut.

- Persiapkan data yang diperlukan, yaitu nama elemen untuk dicari dan jumlah resep yang diminta
- Kemudian dari *map* seluruh elemen, cari *key* yang berupa nama elemen tersebut.
- Jika *key* tidak ditemukan, atau resep elemen tersebut tidak ditemukan, maka elemen tersebut tidak valid, dan fungsi mengembalikan pohon kosong.
- Jika elemen ditemukan sebagai elemen dasar, maka *base case* diraih dan fungsi mengembalikan pohon yang berisi satu *node* yang merupakan leaf dari pohon solusi.
- Jika elemen bukan merupakan elemen dasar, maka resep-resep elemen yang berada pada *map* akan dicek satu-persatu dalam sebuah loop.
- Jika salah satu elemen pada resep memiliki *tier* lebih besar atau sama dengan *tier* elemen yang sedang dicari, maka pasangan resep tersebut akan dilewati.

- Ketika ditemukan pasangan resep yang *valid*, maka akan dilakukan rekursi untuk kedua elemen pada resep secara konkuren. Setelah hasil rekursi didapat, maka akan dicek hasil dari rekursi berisi pohon atau kosong, jika kosong maka resep ini dilewati
- Jika hasil rekursi menghasilkan pohon, selanjutnya di dalam loop yang sama, akan dilakukan kombinasi hasil rekursi.
- Setelah kombinasi, jika jumlah resep yang diminta belum tercapai, maka iterasi pasangan resep selanjutnya akan dilakukan.
- Jika jumlah resep yang diminta sudah tercapai, maka fungsi akan mengembalikan pohon solusi yang telah didapatkan.

3.1.5. Bidirectional

Pencarian resep elemen dengan algoritma bidirectional dilakukan dengan dua pencarian secara simultan: satu dari elemen dasar ke atas (forward) dan satu lagi dari target ke bawah (backward). Langkah-langkahnya adalah sebagai berikut:

- Persiapan parameter awal, yaitu elemen yang dicari , dan jumlah resep yang diinginkan .
- Validasi elemen target. Jika target adalah elemen dasar maka langsung dikembalikan, tetapi jika elemen tidak ditemukan atau tidak punya resep maka dikembalikan hasil kosong.
- Jika target *valid*, pertama dilakukan *Forward Search* (dimulai dari 4 elemen dasar).
- Masing-masing elemen dasar diinisialisasi sebagai pohon node tunggal dan dimasukkan ke dalam *forward queue*—yaitu struktur yang sama untuk BFS, namun untuk empat elemen sekaligus—pada kedalaman 0.
- Selanjutnya, pencarian dilakukan sesuai algoritma BFS, dengan mencoba membentuk elemen-elemen baru dari kombinasi resep yang tersedia pada *map* elemen.
- Jika dua bahan pada suatu resep telah ditemukan dalam lapisan sebelumnya, maka elemen hasilnya dapat dibentuk, dan simpul *tree* baru dibuat dari gabungan kedua child-nya. Hasil tersebut disimpan dalam *forward queue*.
- Setiap elemen yang berhasil dibentuk akan ditambahkan ke antrian forward untuk lapisan berikutnya. Proses ini terus dilakukan bergantian dengan *backward search* hingga kedua arah penelusuran ini menemukan simpul atau elemen yang sama.
- *Backward search* dilakukan serupa dengan *forward search*, namun dimulai dari elemen target, dan yang dicari adalah bahan yang digunakan untuk membuat elemen yang sedang dikunjungi.
- Setelah ditemukan sebuah resep yang valid, maka resep tersebut disimpan, dan akan dicari resep lain hingga jumlah resep yang diinginkan dicapai.

3.2. Frontend

3.2.1 Fitur Fungsional

Aplikasi yang dibuat dirancang untuk menampilkan hasil pencarian elemen secara interaktif dan visual. Berikut adalah fitur-fitur yang tersedia pada web ini.

1. Pencarian Elemen

Semua elemen yang terdapat pada permainan alchemy 2 tersedia pada kolom pencarian. Menu ini dilengkapi dengan dropdown dari data yang valid sehingga memudahkan pengguna dalam mengidentifikasi elemen.

2. Algoritma Pencarian

   Disediakan 3 algoritma yang dapat digunakan pengguna, yaitu BFS (*Breadth First Search*), DFS (Depth First Search), dan bidirectional

3. Input Count

   Jika pengguna hanya ingin melihat 1 resep untuk membuat suatu elemen, maka dapat diisi dengan 1. Namun jika ingin melihat banyak resep untuk mencapainya, pengguna dapat mengisinya sesuai kehendaknya.

4. Fitur Live Update Tree

   Untuk melihat proses pembentukan secara bertahap, pengguna dapat mengaktifkan fitur ini dan tree yang ditampilkan dengan jeda waktu yang dapat diatur oleh pengguna itu sendiri. Fitur ini memanfaatkan websocket, dimana backend menerima parameter dan frontend menerima hasil secara real time, sehingga memungkinkan implementasi fitur ini.

5. Hasil Tree

   Dengan masukkan parameter yang telah ditentukan sebelumnya, website akan berpindah ke halaman baru yang memvisualisasikan hasil pembentukan elemen.

6. Statistik Hasil Pencarian

   Di sebelah kiri hasil tree, akan ditampilkan statistik hasil pencarian yang meliputi total waktu yang ditempuh, dan banyak node (elemen) yang dikunjungi selama proses pencarian.

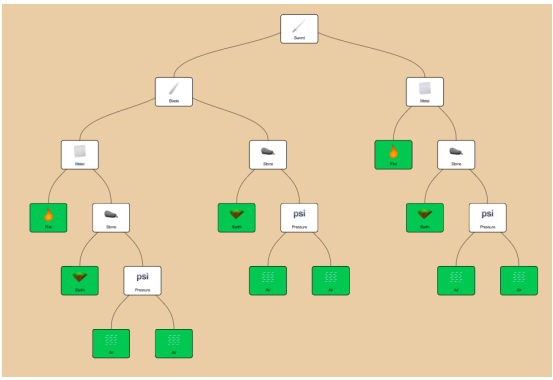
3.3. Contoh Ilustrasi Kasus

Salah satu contoh ilustrasi sederhana penggunaan BFS dan DFS adalah pencarian elemen “Sword” menggunakan elemen-elemen dasar. Berdasarkan website Fandom Alchemy, Sword dibentuk dari 2 elemen anak, yaitu Blade dan Metal. Jika digunakan pencarian BFS, maka untuk setiap anak dari elemen, akan di cek terlebih dahulu berdasarkan urutan level (tingkat kedalaman) sebelum melanjutkan ke level berikutnya. Sedangkan pada pencarian DFS, pencarian dilakukan dengan menelusuri satu cabang hingga mencapai ujung (daun) sebelum kembali ke cabang sebelumnya. Berikut adalah ilustrasi dari kedua algoritma.

1. BFS

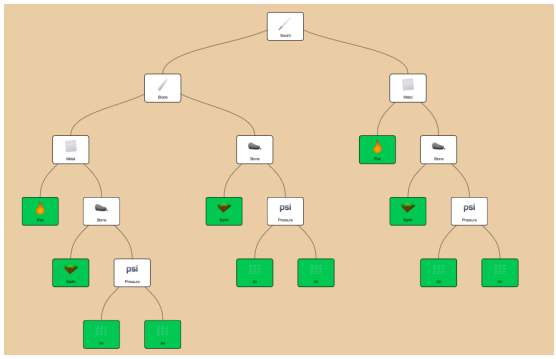
Di mulai dari elemen dasar, gabungkan semua kombinasi di level ini (fire + stone = metal misalnya), kemudian dilanjutkan pada level-level berikutnya hingga sampai di elemen tujuan.

Level 0: Fire, Stone, Air Level 1: Metal (Fire+Stone) Level 2: Blade (Metal+Stone) Level 3: Sword (Blade+Metal)



2. DFS

Berbeda dengan BFS, DFS akan mencoba langsung menelusuri satu jalur hingga menemukan elemen target. Penggunaan DFS akan lebih cepat menemukan Sword jika jalur yang dipilih pertama langsung menuju kombinasi yang benar.



IV. Implementasi dan Testing

4.1. Struktur File

4.1.1 Backend

Berikut adalah struktur file backend yang kami gunakan.

```
.
├── Dockerfile
├── README.md
└── src
    ├── bfs.go
    ├── bidirectional.go
    ├── dfs.go
    ├── go.mod
    ├── go.sum
    ├── main.go
    ├── scrapper.go
    ├── treebuilder.go
    └── data
        └── elements.json
```

4.1.2 Frontend

Berikut adalah struktur file frontend yang kami gunakan.

```
.
├── Dockerfile
├── README.md
├── next.config.ts
├── package.json
├── package-lock.json
├── public
│
└── src
    ├── app
    │   ├── favicon.ico
    │   ├── globals.css
    │   ├── layout.tsx
    │   ├── page.tsx
    │   └── result
```

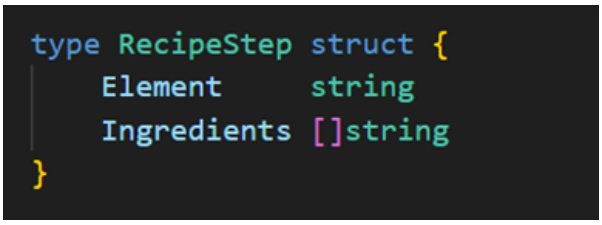
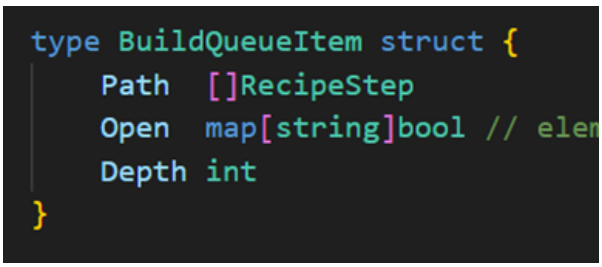
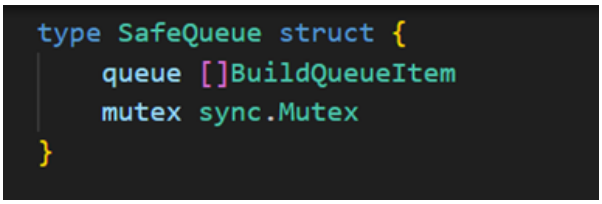
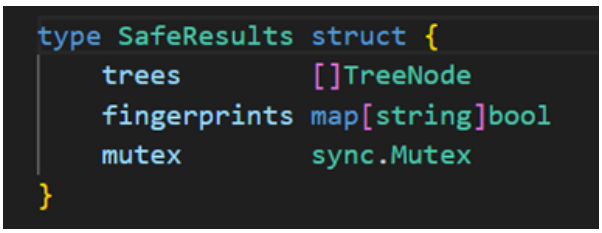
```
│       └── page.tsx
│
├── components
│   ├── ParameterBar
│   │   └── page.tsx
│   ├── ResultBar
│   │   └── page.tsx
│   └── TreeRecipe
│       ├── page.tsx
│       └── TreeClient.tsx
│
├── data
│   ├── basic.json
│   ├── elements.json
│   ├── images.json
│   └── recipe.json
│
└── types
    └── types.tsx
```

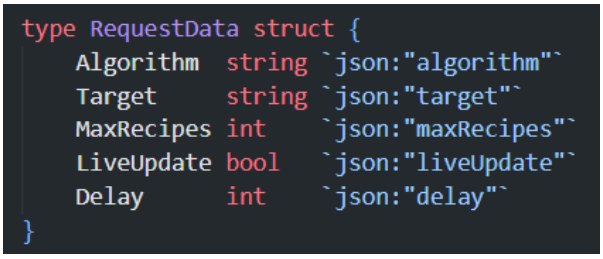
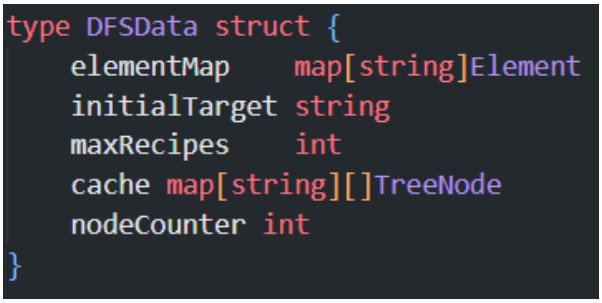
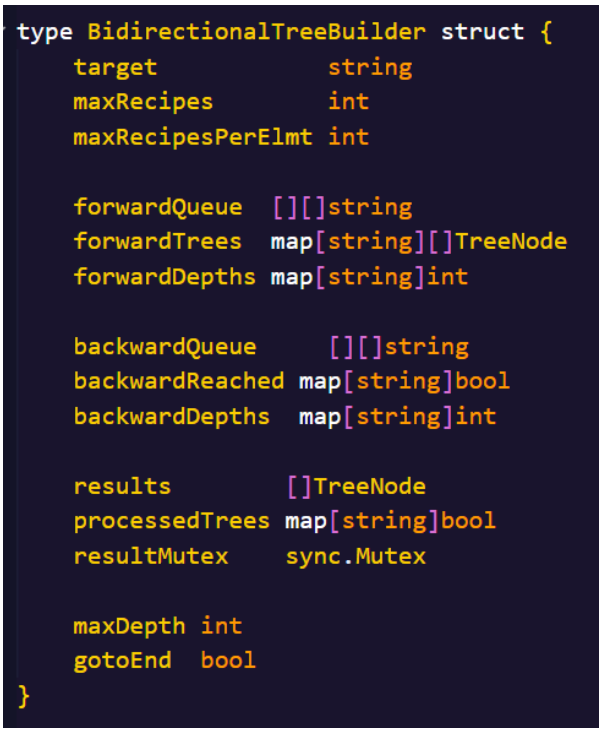
4.2. Struktur Data

4.2.1 Backend

Pada Backend, kami membuat beberapa struktur data yang ditampilkan dalam tabel berikut.

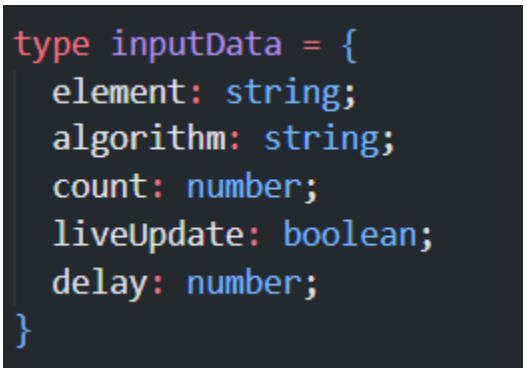
Struktur Data	Penjelasan
`type Element struct { Name string json:"name" Recipes [][]string json:"recipes" Tier int json:"tier" }`	Struktur Element merepresentasikan sebuah entitas atau elemen yang dapat disusun atau dikombinasikan dari elemen lain. Struktur ini digunakan untuk memuat data elemen dari file JSON (misalnya elements.json). Atribut-atributnya meliputi Name, Recipes (array dari nama-nama elemen), dan Tier yang merupakan tingkatan dari elemen.
`type TreeNode struct { Name string json:"name" Children []TreeNode json:"children,omitempty" Highlight bool json:"highlight,omitempty" }`	Struktur TreeNode digunakan untuk membangun representasi pohon dari proses pencarian atau penyusunan elemen. Struktur ini digunakan untuk memudahkan frontend memvisualisasikan tree. Atribut-atributnya meliputi Name, Children (2

	element yang membuat element root), dan highlight sebagai penanda khusus element.
	Struktur RecipeStep digunakan untuk merepresentasikan satu langkah dalam proses pembuatan elemen, yaitu elemen yang dihasilkan beserta dua bahan penyusunnya. Struktur ini digunakan sebagai bagian dari path pencarian dalam BFS dan DFS untuk merekam urutan kombinasi yang dilakukan. Atribut-atributnya adalah Element (nama elemen hasil resep), dan Ingredients (slice string berisi dua nama bahan).
	Struktur BuildQueueItem digunakan untuk menyimpan informasi satu jalur pencarian dalam algoritma BFS. Struktur ini menyimpan urutan langkah (Path) dari target ke elemen dasar, daftar elemen yang masih perlu dicari resepnya (Open), serta kedalaman langkah (Depth) dari pencarian tersebut. Open merupakan map string ke boolean yang menunjukkan elemen mana saja yang belum ditemukan resepnya dan masih perlu diperluas.
	Struktur SafeQueue digunakan sebagai queue thread-safe dalam algoritma BFS multithreaded. Struktur ini menyimpan daftar BuildQueueItem yang akan dieksekusi oleh worker thread. Untuk menjaga konsistensi data saat diakses oleh banyak thread, SafeQueue menggunakan mutex sebagai pengunci akses saat melakukan push dan pop. Struktur ini memungkinkan sistem antrian berjalan secara paralel tanpa race condition.
	Struktur SafeResults digunakan untuk menyimpan hasil pencarian berupa solusi pohon resep (TreeNode). Selain menyimpan hasil (trees), struktur ini juga menyimpan fingerprints berupa string unik dari struktur pohon untuk mencegah duplikasi hasil. SafeResults bersifat thread-safe dan menggunakan mutex agar penambahan hasil oleh banyak thread tetap aman. Dengan struktur ini, hasil akhir pencarian BFS dapat dikoleksi secara efisien dan bebas duplikat.

	Struktur data yang digunakan untuk menerima JSON parameter hasil kiriman dari frontend. RequestData memiliki nama dan jenis atribut sesuai dengan nama atribut dalam JSON parameter. Algorithm adalah algoritma yang dipakai (BFS/DFS), target adalah element yang dicari, MaxRecipes adalah recipe maksimal yang di input user, LiveUpdate merupakan boolean jika user ingin menampilkan tree secara live update, dan delay adalah waktu delay tiap tree update.
	Struktur DFSData digunakan untuk menyimpan data internal dan konteks selama proses pencarian dengan algoritma Depth-First Search (DFS). Struktur ini menyederhanakan pemrosesan dan memungkinkan pengelolaan status serta caching hasil secara efisien (caching khusus untuk live update). Atribut-atributnya adalah , initialTarget (nama elemen tujuan), maxRecipes (banyak recipe yang ingin diperoleh), cache (memoisasi *tree recipe*), dan nodeCounter (menghitung banyak node yang telah dikunjungi).
	Struktur BIDTreeData digunakan untuk menyimpan data internal dan konteks selama proses pencarian dengan algoritma Bidirectional Search (BFS dua arah). Struktur ini mempermudah implementasi *bidirectional search*, dengan menyimpan data-data yang penting. Atribut-atribut penting di dalamnya mencakup target (elemen akhir yang ingin dicapai), maxRecipes dan maxRecipesPerElmt (batas jumlah resep yang dicari secara keseluruhan dan per elemen), serta *forward* dan *backward queue*, dan penyimpanan pohon hasil pencarian. processedTrees digunakan untuk mentimpan hasil *tree* setelah di-deduplikasi. Atribut resultMutex digunakan untuk menjaga keamanan akses data saat multi-threading, sementara maxDepth, gotoEnd, dan nodesVisited membantu mengatur batas pencarian dan mengukur performa proses.

4.2.2 Frontend

Pada Frontend, kami membuat satu struktur data (*types*) yang ditampilkan dalam tabel berikut.

Struktur Data	Penjelasan
	Tipe inputData adalah struktur data dalam TypeScript yang digunakan untuk merepresentasikan input dari pengguna pada sisi frontend, yang nantinya akan dikirim ke backend sebagai parameter pencarian elemen. Algorithm adalah algoritma yang dipakai (BFS/DFS), target adalah element yang dicari, MaxRecipes adalah recipe maksimal yang di input user, LiveUpdate merupakan boolean jika user ingin menampilkan tree secara live update, dan delay adalah waktu delay tiap tree update.

4.3. Fungsi

4.3.1 Backend

Pada Backend, kami membuat beberapa fungsi yang digunakan sebagai pondasi utama jalannya metode BFS dan DFS.

Fungsi bfsMultiple
Fungsi ini digunakan sebagai *interface* fungsi *main* untuk algoritma BFS, yaitu sebagai fungsi yang memanggil algoritma BFS utama dan menampung hasilnya. Fungsi inilah yang akan dipanggil pada fungsi *main*.

`func bfsMultiple(elementMap map[string]Element, target string, maxRecipes int) ([]TreeNode, int) {`
`  target = strings.ToLower(target)`
 	
`  counter := &Counter{}`
 	
`  if isBasicElement(target) {`
`  return []TreeNode{{Name: capitalize(target)}}, counter.Get()`
`  }`
 	
`  if elem, ok := elementMap[target]; !ok || len(elem.Recipes) == 0 {`
`  return []TreeNode{}, counter.Get()`

```
	}

	trees := bfsBuildRecipeTreesParallel(target, elementMap, maxRecipes, counter)
	fmt.Printf("Total nodes visited: %d\n", counter.Get())
	return trees, counter.Get()
}
```

Fungsi bfsBuildRecipeTreesParallel

Fungsi ini yang akan di panggil oleh bfsMultiple untuk membantu dalam melakukan pencarian.

```
func bfsBuildRecipeTreesParallel(target string, elementMap map[string]Element,
maxRecipes int, counter *Counter) []TreeNode {
	queue := newSafeQueue()
	results := newSafeResults(maxRecipes)
	pathKeys := newSafePathKeys()

	// Inisialisasi queue dengan recipe awal
	queue.Push(createInitialQueueItems(target, elementMap)...)

	var wg sync.WaitGroup
	done := make(chan struct{})

	// Membuat worker goroutines
	for i := 0; i < numWorkers; i++ {
		wg.Add(1)
		go worker(queue, results, pathKeys, elementMap, target, counter, &wg,
done)
	}

	// Goroutine untuk memonitor kondisi selesai
	go func() {
		for {
			time.Sleep(100 * time.Millisecond)
			if queue.Length() == 0 || results.IsFull() {
				close(done) // Signal all workers to finish
				break
			}
		}
	}()

	wg.Wait()
	return results.GetTrees()
}
```

Fungsi bfsMultipleLive

Fungsi ini sama seperti bfsMultiple, namun digunakan untuk fitur *live update*, dimana algoritma utama BFS khusus untuk *live update* akan melakukan pengiriman data *tree* kepada *web* seiring penelusuran

```
func bfsMultipleLive(elementMap map[string]Element, target string, maxRecipes int,
delay int, conn *websocket.Conn) []TreeNode {
    target = strings.ToLower(target)
    counter := &Counter{}
    pathKeys := newSafePathKeys()
    results := newSafeResults(maxRecipes)

    if isBasicElement(target) {
        return []TreeNode{{Name: capitalize(target)}}
    }

    if elem, ok := elementMap[target]; !ok || len(elem.Recipes) == 0 {
        return []TreeNode{}
    }

    queue := newSafeQueue()
    queue.Push(createInitialQueueItems(target, elementMap)...)

    conn.WriteJSON(map[string]interface{}{
        "status":       "Starting",
        "message":      "Initializing single-threaded BFS...",
        "treeData":     []TreeNode{},
        "nodesVisited": 0,
    })

    previewSent := make(map[string]bool)

    for queue.Length() > 0 && !results.IsFull() {
        items := queue.Pop(1)
        if len(items) == 0 {
            break
        }
        curr := items[0]
        counter.Increment()

        if len(curr.Path) > 0 {
            lastStep := curr.Path[len(curr.Path)-1]
            previewKey := pathToStringKey([]RecipeStep{lastStep})
            if !previewSent[previewKey] {
                tree := buildTreeFromSteps(lastStep.Element, []RecipeStep{lastStep},
elementMap)
                conn.WriteJSON(map[string]interface{}{
                    "status":       "Preview",
                    "message":      "Discovered: " + capitalize(lastStep.Element),
                    "treeData":     []TreeNode{tree},
                    "nodesVisited": counter.Get(),
                })
                time.Sleep(time.Duration(delay) * time.Millisecond)
                previewSent[previewKey] = true
            }
        }

        if curr.Depth > bfsMaxDepth {
            continue
```

```
        }

        if len(curr.Open) == 0 {
            key := pathToStringKey(curr.Path)
            if pathKeys.Check(key) || isStructuralDuplicate(curr.Path, elementMap,
pathKeys) {
                continue
            }
            pathKeys.Add(key)

            fp := canonicalizeSteps(curr.Path, elementMap)
            if !results.Add(TreeNode{}, fp) {
                continue
            }

            tree := buildTreeFromSteps(target, curr.Path, elementMap)
            results.trees[len(results.trees)-1] = tree

            conn.WriteJSON(map[string]interface{}{
                "status":       "Final",
                "message":      "Final tree found!",
                "treeData":     []TreeNode{tree},
                "nodesVisited": counter.Get(),
            })
            time.Sleep(time.Duration(delay) * time.Millisecond)
            continue
        }

        for openElem := range curr.Open {
            queue.Push(expandOpenElement(openElem, curr, elementMap)...)
            break
        }

        queue.PruneLargeWithPriority()
    }

    conn.WriteJSON(map[string]interface{}{
        "status":       "Completed",
        "message":      fmt.Sprintf("Found %d recipes, explored %d nodes",
results.Count(), counter.Get()),
        "nodesVisited": counter.Get(),
    })

    return results.GetTrees()
}
```

Fungsi dfsMultiple

Fungsi ini digunakan sebagai *interface* fungsi *main* untuk algoritma DFS, yang bertanggung jawab mempersiapkan struktur data yang digunakan untuk dfsRecursive, serta bertanggung jawab memanggil dan menampung hasil dari fungsi dfsRecursive.

```
func dfsMultiple(target string, maxRecipes int) ([]TreeNode, int) {
	AlgoData := AlgoData{
		initialTarget: strings.ToLower(target),
		maxRecipes:    maxRecipes,
		nodeCounter:   0,
	}

	var resultTrees []TreeNode
	resultTrees = AlgoData.dfsRecursiveMultithread(strings.ToLower(target))

	if maxRecipes > 0 && len(resultTrees) > maxRecipes {
		return resultTrees[:maxRecipes], int(AlgoData.nodeCounter)
	}
	return resultTrees, int(AlgoData.nodeCounter)
}
```

Fungsi dfsRecursive

Fungsi ini adalah fungsi utama dari algoritma DFS, yang melakukan penelusuran graf elemen serta menyimpan pohon-pohon solusi dari resep elemen yang ingin dicari

```
func (d *AlgoData) dfsRecursive(currElement string) []TreeNode {
	atomic.AddInt64(&d.nodeCounter, 1)
	currElement = strings.ToLower(currElement)

	elemDetails, exists := elementMap[currElement]
	if !exists {
		return []TreeNode{}
	}

	if isBasicElement(elemDetails.Name) {
		leafNode := TreeNode{Name: elemDetails.Name}
		basicTreeList := []TreeNode{leafNode}
		return basicTreeList
	}

	if len(elemDetails.Recipes) == 0 {
		leafNode := TreeNode{Name: elemDetails.Name}
		noRecipeTreeList := []TreeNode{leafNode}
		return noRecipeTreeList
	}

	currTreeCombinations := make([]TreeNode, 0)
	productTier := elemDetails.Tier

	recipePairLoop:
	for _, recipePair := range elemDetails.Recipes {
		if len(recipePair) != 2 {
			continue
		}
		parent1Name := strings.ToLower(recipePair[0])
		parent2Name := strings.ToLower(recipePair[1])
```

```
            elemParent1, p1Exists := elementMap[parent1Name]
            elemParent2, p2Exists := elementMap[parent2Name]

            if !p1Exists || !p2Exists {
                  continue
            }
            if elemParent1.Tier >= productTier || elemParent2.Tier >= productTier {
                  continue
            }

            var subTreesForParent1 []TreeNode
            var subTreesForParent2 []TreeNode
            var wg sync.WaitGroup
            wg.Add(2)

            go func() {
                  defer wg.Done()
                  subTreesForParent1 = d.dfsRecursive(parent1Name)
            }()

            go func() {
                  defer wg.Done()
                  subTreesForParent2 = d.dfsRecursive(parent2Name)
            }()

            wg.Wait()
            if !isBasicElement(elemParent1.Name) && len(subTreesForParent1) == 0
{continue}
            if !isBasicElement(elemParent2.Name) && len(subTreesForParent2) == 0
{continue}

            combinationLoop:
            for _, treeP1 := range subTreesForParent1 {
                  for _, treeP2 := range subTreesForParent2 {
                        if d.maxRecipes > 0 && len(currTreeCombinations) >=
d.maxRecipes {
                              break combinationLoop
                        }

                        newNode := TreeNode{
                              Name:     elemDetails.Name,
                              Children: []TreeNode{treeP1, treeP2},
                        }
                        currTreeCombinations = append(currTreeCombinations,
newNode)
                  }
            }

            if d.maxRecipes > 0 && len(currTreeCombinations) >= d.maxRecipes {
                  break recipePairLoop
            }
      }
      return currTreeCombinations
}
```

Fungsi dfsMultipleLive

Sama seperti bfsMultipleLive, fungsi ini merupakan variasi DFS untuk *live update*

```
func dfsMultipleLive(elementMap map[string]Element, target string, maxRecipes int,
delay int, conn *websocket.Conn) ([]TreeNode, int) {
	dfsData := DFSData{
		elementMap:    elementMap,
		initialTarget: strings.ToLower(target),
		maxRecipes:    maxRecipes,
		cache:         make(map[string][]TreeNode), // Cache stores []TreeNode
		nodeCounter: 0,
	}

	resultTreeNodes := dfsData.dfsRecursiveLive(strings.ToLower(target), delay,
conn)

	if maxRecipes > 0 && len(resultTreeNodes) > maxRecipes {
		return resultTreeNodes[:maxRecipes], dfsData.nodeCounter
	}
	return resultTreeNodes, dfsData.nodeCounter
}
```

Fungsi dfsRecursiveLive

Fungsi utama algoritma DFS yang menelusuri, menyimpan serta mengirimkan pohon-pohon solusi kepada *web* untuk penampilan *live update*

```
func (d *DFSData) dfsRecursiveLive(elementToMakeCurrently string, delay int, conn
*websocket.Conn) []TreeNode {
	d.nodeCounter++
	elementToMakeCurrently = strings.ToLower(elementToMakeCurrently)

	if cachedResult, found := d.cache[elementToMakeCurrently]; found {
		return cachedResult
	}

	elemDetails, exists := d.elementMap[elementToMakeCurrently]
	if !exists {
		d.cache[elementToMakeCurrently] = []TreeNode{}
		return []TreeNode{}
	}

	currentElementNameFormatted := elemDetails.Name


	if isBasicElement(elemDetails.Name) {
		leafNode := TreeNode{Name: currentElementNameFormatted}
		basicTreeList := []TreeNode{leafNode}
		d.cache[elementToMakeCurrently] = basicTreeList
		return basicTreeList
	}
```

```
	if len(elemDetails.Recipes) == 0 {
		leafNode := TreeNode{Name: currentElementNameFormatted}
		noRecipeTreeList := []TreeNode{leafNode}
		d.cache[elementToMakeCurrently] = noRecipeTreeList
		return noRecipeTreeList
	}

	var operationalLimit int
	isInitialTarget := (d.initialTarget == elementToMakeCurrently)

	if d.maxRecipes <= 0 {
		operationalLimit = 0
	} else if isInitialTarget {
		operationalLimit = d.maxRecipes
	} else {
		if d.maxRecipes < 10 {
			operationalLimit = 20
		} else {
			operationalLimit = d.maxRecipes
		}
	}

	allPossibleTreesForCurrentElement := make([]TreeNode, 0)
	productTier := elemDetails.Tier

recipePairLoop:
	for _, recipePair := range elemDetails.Recipes {
		if len(recipePair) != 2 {
			continue
		}
		parent1Name := strings.ToLower(recipePair[0])
		parent2Name := strings.ToLower(recipePair[1])

		elemParent1, p1Exists := d.elementMap[parent1Name]
		elemParent2, p2Exists := d.elementMap[parent2Name]

		if !p1Exists || !p2Exists {
			continue
		}
		if elemParent1.Tier >= productTier || elemParent2.Tier >= productTier {
			continue
		}
		if strings.Contains(elemParent1.Name, "fanon") ||
strings.Contains(elemParent2.Name, "fanon") {
			continue
		}

		subTreesForParent1 := d.dfsRecursiveLive(parent1Name, delay, conn)
		if !isBasicElement(elemParent1.Name) && len(subTreesForParent1) == 0 {
			continue
		}

		subTreesForParent2 := d.dfsRecursiveLive(parent2Name, delay, conn)
```

```
            if !isBasicElement(elemParent2.Name) && len(subTreesForParent2) == 0 {
                  continue
            }

      combinationLoop:
            for _, treeP1 := range subTreesForParent1 {
                  for _, treeP2 := range subTreesForParent2 {
                        if operationalLimit > 0 &&
len(allPossibleTreesForCurrentElement) >= operationalLimit {
                              break combinationLoop
                        }

                        newNode := TreeNode{
                              Name:     currentElementNameFormatted,
                              Children: []TreeNode{treeP1, treeP2},
                        }
                        allPossibleTreesForCurrentElement =
append(allPossibleTreesForCurrentElement, newNode)
                  }
            }

            if operationalLimit > 0 && len(allPossibleTreesForCurrentElement) >=
operationalLimit {
                  break recipePairLoop
            }
      }

      conn.WriteJSON(map[string]interface{}{
            "status":   "Progress",
            "message":  "Finding " + currentElementNameFormatted + " trees",
            "duration": 0,
            "treeData": allPossibleTreesForCurrentElement,
            "nodesVisited": d.nodeCounter,
      })
      time.Sleep(time.Duration(delay) * time.Millisecond)

      d.cache[elementToMakeCurrently] = allPossibleTreesForCurrentElement
      return allPossibleTreesForCurrentElement
}
```

Fungsi bidirectionalMultiple

Fungsi ini digunakan sebagai *interface* fungsi *main* untuk algoritma Bidirectional, yang bertanggung jawab mempersiapkan struktur data yang digunakan untuk algoritma Bidirectional

```
func bidirectionalMultiple(target string, maxRecipes int, maxRecipesPerElmt int)
([]TreeNode, int) {
      targetLower := strings.ToLower(target)

      if isBasicElement(targetLower) {
            return []TreeNode{{Name: capitalize(targetLower)}}, 1
```

```
	}
	targetElem, exists := elementMap[targetLower]
	if !exists || len(targetElem.Recipes) == 0 {
		return []TreeNode{}, 0
	}

	BIDData := &BIDTreeData{
		target:            targetLower,
		maxRecipes:        maxRecipes,
		maxRecipesPerElmt: maxRecipesPerElmt,
		forwardQueue:      make([][]string, 1),
		forwardTrees:      make(map[string][]TreeNode),
		forwardDepths:     make(map[string]int),
		backwardQueue:     make([][]string, 1),
		backwardReached:   make(map[string]bool),
		backwardDepths:    make(map[string]int),
		results:           make([]TreeNode, 0),
		processedTrees:    make(map[string]bool),
		maxDepth:          20,
		gotoEnd:           false,
	}

	initializeForwardSearch(BIDData, elementMap)
	initializeBackwardSearch(BIDData)

	fLayer, bLayer := 0, 0
	for fLayer < BIDData.maxDepth && bLayer < BIDData.maxDepth {
		if BIDData.gotoEnd {
			break
		}

		forwardExpanded := expandForwardLayer(BIDData, fLayer, elementMap)
		if forwardExpanded {
			fLayer++
		}

		if BIDData.gotoEnd {
			break
		}

		backwardExpanded := expandBackwardLayer(BIDData, bLayer, elementMap)
		if backwardExpanded {
			bLayer++
		}

		if !forwardExpanded && !backwardExpanded &&
		   (fLayer >= len(BIDData.forwardQueue) ||
len(BIDData.forwardQueue[fLayer]) == 0) &&
		   (bLayer >= len(BIDData.backwardQueue) ||
len(BIDData.backwardQueue[bLayer]) == 0) {
			break
		}
	}
```

```
	return BIDData.results, int(BIDData.nodesVisited)
}
```

Fungsi handleWebSocket

Fungsi ini digunakan untuk menghidupkan *web socket*, yang merupakan penghubung algoritma *backend* dengan bagian *frontend* dari web. Fungsi inilah yang akan menerima data input user dari aplikasi *web*, dan meneruskannya untuk pemanggilan algoritma-algoritma sesuai *input* dari *user*

```
func handleWebSocket(w http.ResponseWriter, r *http.Request) {
	conn, err := upgrader.Upgrade(w, r, nil)
	if err != nil {
		log.Println("Upgrade error:", err)
		return
	}
	defer conn.Close()

	log.Println("WebSocket connection established")

	var reqData RequestData
	err = conn.ReadJSON(&reqData)
	if err != nil {
		log.Println("Read error:", err)
		return
	}

	log.Printf("Received request - Element: %s, Algorithm: %s, MaxRecipes: %s",
		reqData.Target, reqData.Algorithm, reqData.MaxRecipes)

	conn.WriteJSON(map[string]interface{}{
		"status":  "Processing",
		"message": "Loading elements data",
	})

	data, err := ioutil.ReadFile("data/elements.json")
	if err != nil {
		log.Fatalf("Failed to read elements.json: %v", err)
		conn.WriteJSON(map[string]interface{}{
			"error": "Failed to read elements data",
		})
		return
	}

	var elements []Element
	if err := json.Unmarshal(data, &elements); err != nil {
		log.Fatalf("Failed to parse JSON: %v", err)
		conn.WriteJSON(map[string]interface{}{
			"error": "Failed to parse elements data",
		})
		return
	}
```

```
	elementMap := make(map[string]Element)
	for _, e := range elements {
		elementMap[strings.ToLower(e.Name)] = e
	}

	var recipePlans []TreeNode

	maxRecipeInput, err := strconv.Atoi(reqData.MaxRecipes)
	if err != nil {
		conn.WriteJSON(map[string]interface{}{
			"error": "Invalid MaxRecipes value",
		})
		return
	}

	var nodesVisited int
	startTime := time.Now()
	fmt.Printf("Delay: %d\n", reqData.Delay)

	if reqData.Algorithm == "BFS" {

		conn.WriteJSON(map[string]interface{}{
			"status":  "Starting BFS",
			"message": "Initializing search algorithm",
		})

		if reqData.LiveUpdate {
			recipePlans = bfsMultipleLive(elementMap,
strings.ToLower(reqData.Target), maxRecipeInput, reqData.Delay, conn)
			log.Println("BFS Live Update")
		} else {
			recipePlans = bfsMultiple(elementMap,
strings.ToLower(reqData.Target), maxRecipeInput)
		}

	} else if reqData.Algorithm == "DFS" {

		conn.WriteJSON(map[string]interface{}{
			"status":  "Starting DFS",
			"message": "Initializing search algorithm",
		})

		if reqData.LiveUpdate {
			recipePlans, nodesVisited = dfsMultipleLive(elementMap,
strings.ToLower(reqData.Target), maxRecipeInput, reqData.Delay, conn)
		} else {
			recipePlans, nodesVisited = dfsMultiple(elementMap,
strings.ToLower(reqData.Target), maxRecipeInput)
		}
	} else if reqData.Algorithm == "BID" {

		conn.WriteJSON(map[string]interface{}{
			"status":  "Starting Bidirectional Search",
```

```
                        "message": "Initializing search algorithm",
                })

                if reqData.LiveUpdate {
                        recipePlans = bidirectionalSearchLive(elementMap,
strings.ToLower(reqData.Target), maxRecipeInput, reqData.Delay, conn)
                } else {
                        recipePlans = bidirectionalSearch(elementMap,
strings.ToLower(reqData.Target), maxRecipeInput)
                }
        }

        elapsed := time.Since(startTime)
        fmt.Printf("Ditemukan %d resep via %s.\n", len(recipePlans),
reqData.Algorithm)

        if len(recipePlans) == 0 {
                conn.WriteJSON(map[string]interface{}{
                        "status":  "Completed",
                        "message": "No recipe plans found",
                })
                return
        }

        fmt.Println("Waktu eksekusi: ", elapsed)

        conn.WriteJSON(map[string]interface{}{
                "status":   "Completed",
                "message":  fmt.Sprintf("Found %d recipe plans", len(recipePlans)),
                "duration": elapsed.String(),
                "treeData": recipePlans,
                "nodes": nodesVisited,
        })
}
```

Fungsi main

Fungsi utama, yang menjadi penghubung *frontend* dan *backend*

```
func main() {
        http.HandleFunc("/ws", handleWebSocket)

        http.HandleFunc("/data", func(w http.ResponseWriter, r *http.Request) {
                w.Header().Set("Access-Control-Allow-Origin", "http://localhost:3000")
                w.Header().Set("Access-Control-Allow-Methods", "GET")
                w.Header().Set("Access-Control-Allow-Headers", "Content-Type")
                w.Header().Set("Content-Type", "application/json")
                http.ServeFile(w, r, "../public/tree.json")
        })

        log.Println("Server started at http://localhost:8080")
        log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Fungsi Scraping

Melakukan *scraping* data dari *website fandom* untuk menjadi sumber data-data yang akan diolah aplikasi

```
func Scraping() {
	baseURL := "https://little-alchemy.fandom.com"
	listURL := baseURL + "/wiki/Elements_(Little_Alchemy_2)"

	fmt.Println("Starting Little Alchemy 2 recipe scraper...")

	elementsList, err := getElementsList(listURL)
	if err != nil {
		log.Fatalf("Failed to get elements list: %v", err)
	}

	seen := map[string]bool{}
	for _, n := range elementsList {
		seen[strings.ToLower(n)] = true
	}
	for _, b := range basicElements {
		if !seen[b] {
			elementsList = append(elementsList, b)
			seen[b] = true
		}
	}

	fmt.Printf("Found %d elements to scrape\n", len(elementsList))
	if len(elementsList) > 0 {
		fmt.Println("Sample:", elementsList[:min(5, len(elementsList))])
	}

	var elements []Element
	var wg sync.WaitGroup
	sem := make(chan struct{}, 3)
	var mu sync.Mutex

	for _, name := range elementsList {
		wg.Add(1)
		go func(name string) {
			defer wg.Done()
			sem <- struct{}{}
			defer func() { <-sem }()

			url := baseURL + "/wiki/" + strings.ReplaceAll(name, " ", "_")
			fmt.Printf("Scraping: %s\n", name)
			recs, err := scrapeRecipesLA2Only(url, name)
			if err != nil {
				log.Printf("  error on %s: %v\n", name, err)
				return
			}

			normalizedRecs := normalizeRecipes(recs)
```

```
                        mu.Lock()
                        elements = append(elements, Element{
                                Name:    name,
                                Recipes: normalizedRecs,
                                Tier:    -1,
                        })
                        mu.Unlock()

                        time.Sleep(300 * time.Millisecond)
                }(name)
        }
        wg.Wait()

        elementsWithNoRecipes := 0
        for _, e := range elements {
                if len(e.Recipes) == 0 && !contains(basicElements,
strings.ToLower(e.Name)) {
                        fmt.Printf("WARNING: %s has no recipes\n", e.Name)
                        elementsWithNoRecipes++
                }
        }
        fmt.Printf("Elements with no recipes: %d\n", elementsWithNoRecipes)

        fmt.Println("Basic elements status:")
        for _, b := range basicElements {
                found := false
                for _, e := range elements {
                        if strings.ToLower(e.Name) == strings.ToLower(b) {
                                found = true
                                break
                        }
                }
                fmt.Printf("%s: %v\n", b, found)
        }

        calcTiersFix(elements)

        outFile := "elements.json"
        if err := saveJSON(elements, outFile); err != nil {
                log.Fatalf("Failed saving %s: %v", outFile, err)
        }
        fmt.Printf("Done! Data with tiers in %s\n", outFile)

        analyzeTiers(elements)
}
```

4.4. Tata Cara Penggunaan

1. Jika menggunakan website, maka langsung dapat mengakses link berikut https://alchemy-bolang.vercel.app/ . Jika ingin menggunakan docker, maka clone repository FE dan BE yang terdapat pada lampiran, dan pastikan kedua folder tersebut berada dalam root yang

sama. Di root directory yang sama, buat file baru bernama docker-compose.yml yang berisi seperti berikut.

docker-compose.yml

```
version: "3.8"

services:
  frontend:
    build:
      context: ./Tubes2_FE_Bolang
    ports:
      - "3000:3000"
    environment:
      - NEXT_TELEMETRY_DISABLED=1
      - NEXT_PUBLIC_BACKEND_WS=ws://backend:8080/ws
    depends_on:
      - backend
    networks:
      - app-network

  backend:
    build:
      context: ./Tubes2_BE_Bolang
    ports:
      - "8080:8080"
    networks:
      - app-network

networks:
  app-network:
    driver: bridge
```

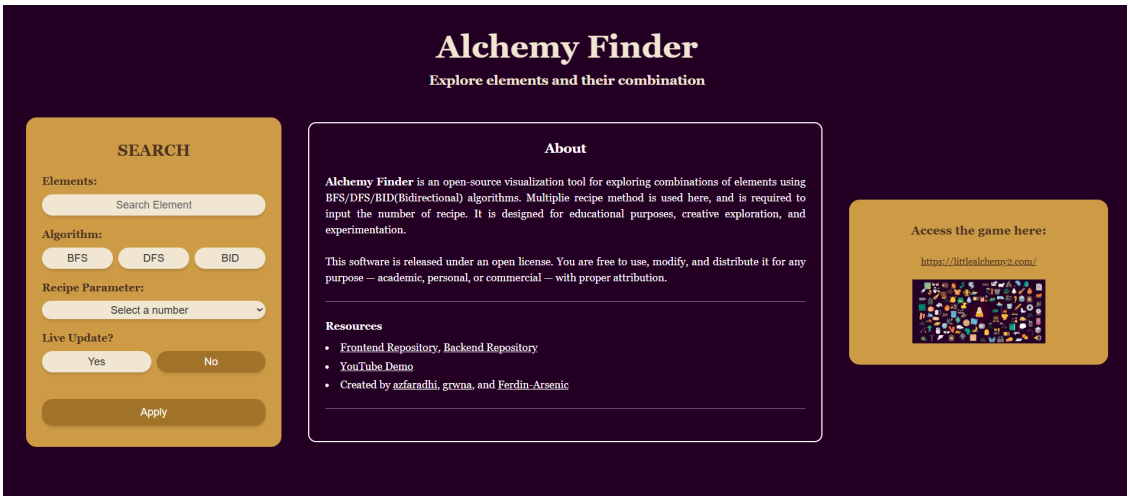
Kemudian tambahkan file .env berikut ini pada root repository Tubes2_FE_Bolang sebagai penghubung antara backend dan frontend. Dari directory root, masukkan command `docker compose up` dan website dapat diakses melalui localhost:3000

.env

```
NEXT_PUBLIC_ENVIRONMENT=local
NEXT_PUBLIC_LOCAL_WEBSOCKET_URL=ws://localhost:8080/ws
```

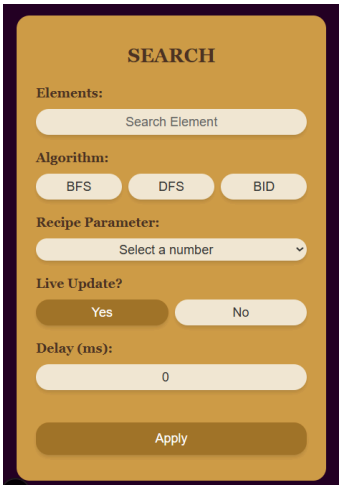
Berikut adalah tampilan awal website.

Tampilan awal website



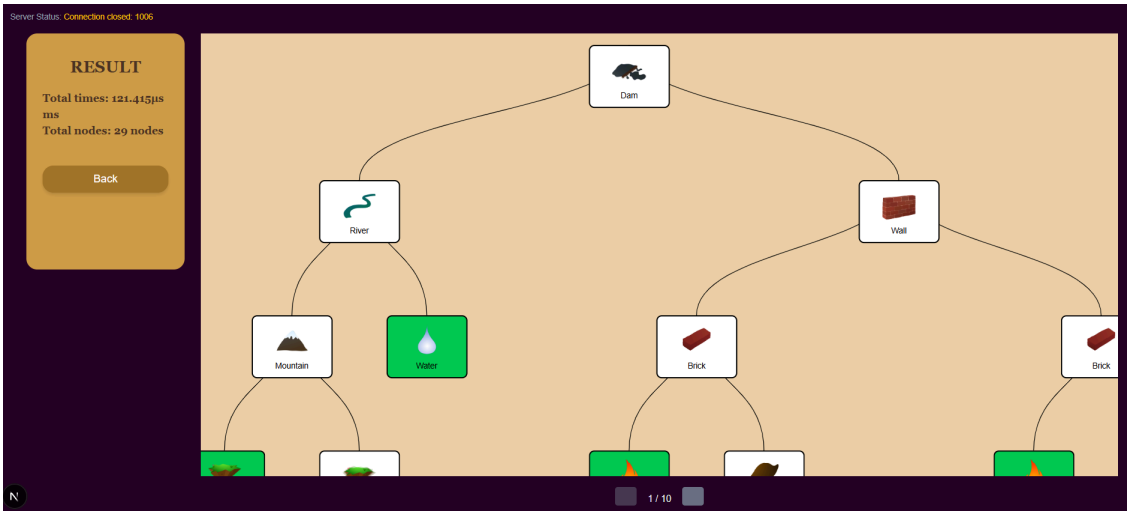
2. Pada kolom search, terdapat beberapa parameter yang dibutuhkan untuk mencari recipe dari suatu elemen. Masukkan nama element, algoritma, dan recipe parameter yang diinginkan. Jika ingin menggunakan fitur live update, maka akan muncul input untuk memasukkan delay tiap tree ditampilkan.

Input Delay



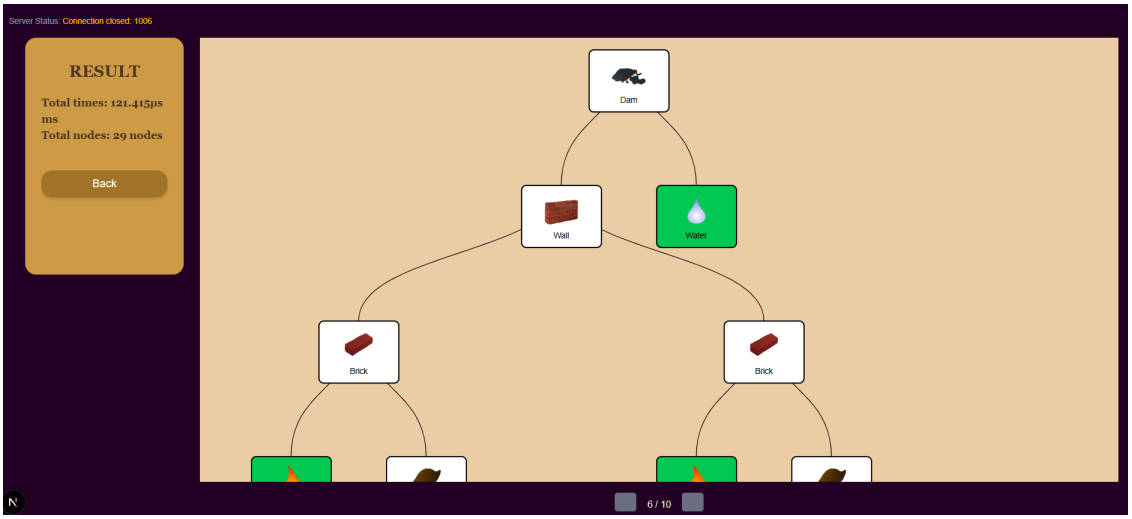
3. Ketika semua parameter telah diisi, tekan apply dan akan diarahkan ke sebuah page baru. Jika memilih tanpa live update, maka akan langsung ditampilkan hasil dari pencarian element sesuai metode yang dipilih.

Tampilan tanpa live tree



Di bagian bawah, terdapat pagination yang memungkinkan user untuk melihat hasil recipe yang lain.

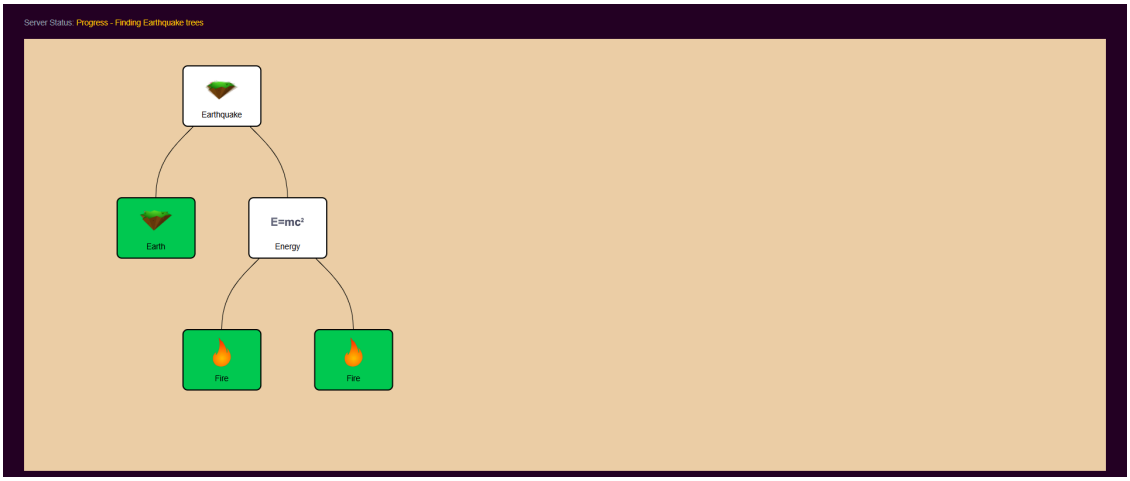
Hasil recipe yang lain



Bar Result menampilkan hasil dari pencarian, berupa waktu yang dibutuhkan dan total node yang dikunjungi.

4. Berikut adalah tampilan dari live update tree.

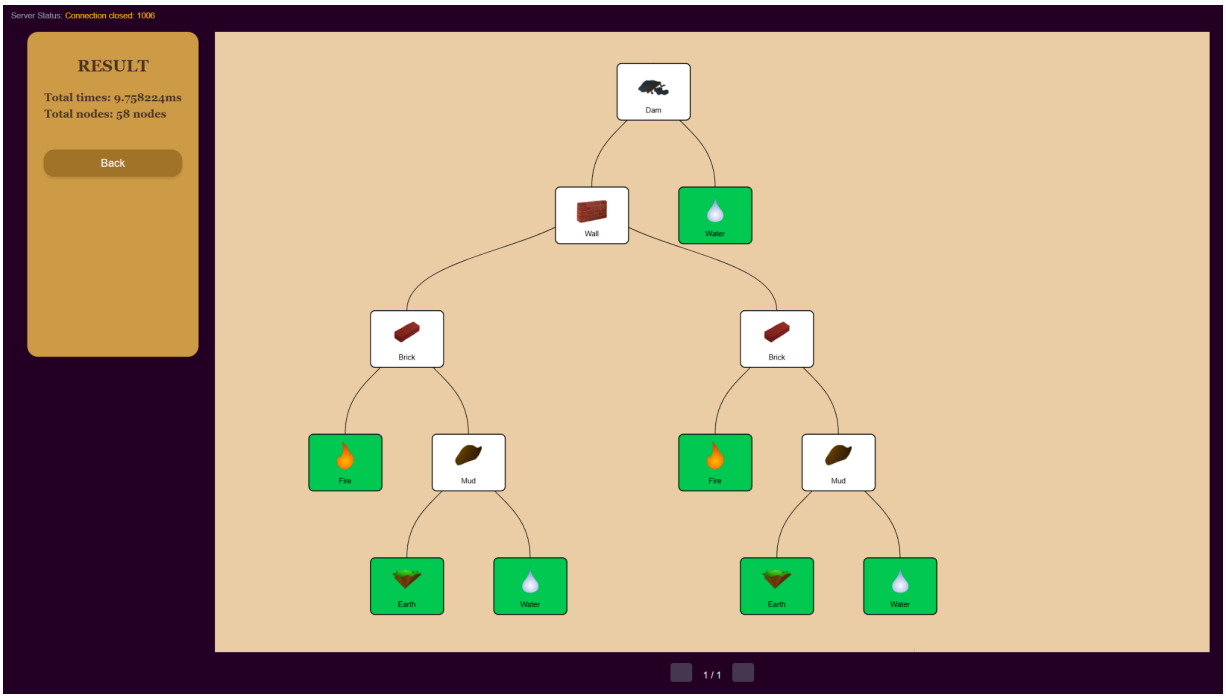
Tampilan Live Tree



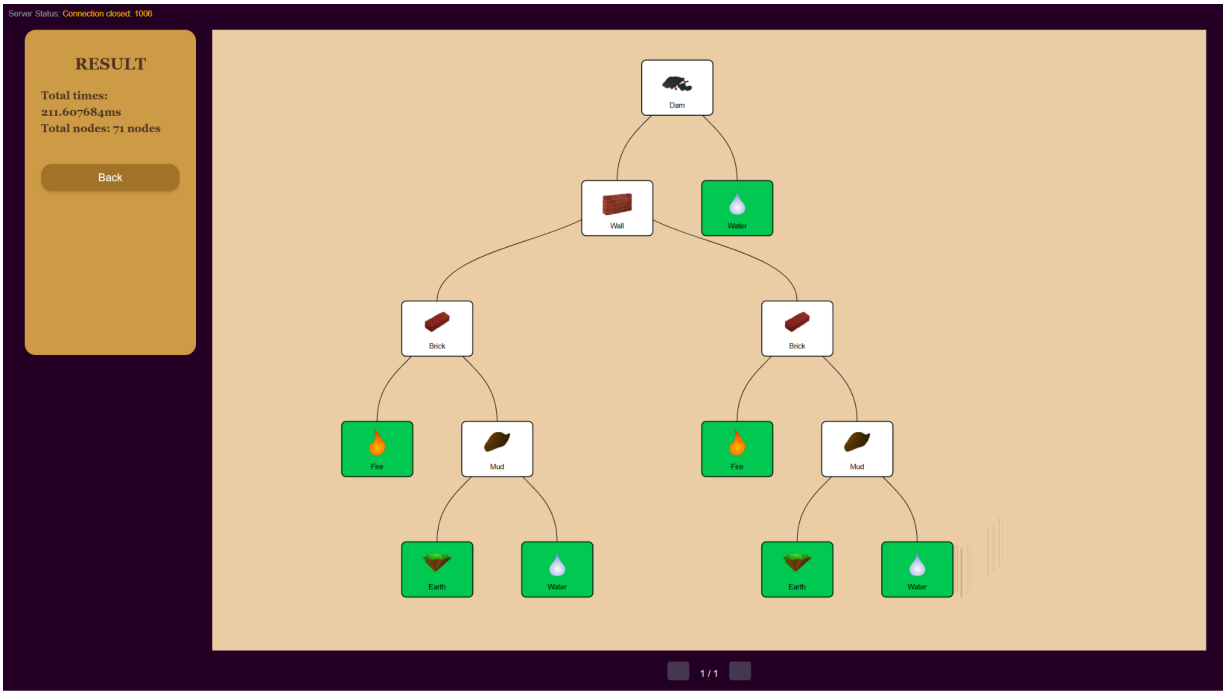
Dalam setiap prosesnya, akan dimunculkan status pencarian yang saat ini sedang ditelusuri. Hasil akhir yang ditampilkan sama seperti jika tanpa menggunakan fitur live.

4.5. Analisis Hasil Pengujian

Berikut akan kami uji program buatan kami berdasarkan elemen, algoritma, dan banyak recipe yang dicari.

Element: Dam, Algorithm: DFS, Count: 1
Hasil:    **Analisis:** Dam berhasil ditemukan menggunakan algoritma DFS dalam waktu sekitar 9.75 milisecond dengan total 58 node yang telah dikunjungi selama proses pencarian berlangsung. Hasil yang didapat menampilkan 1 kombinasi resep yang memungkinkan untuk elemen Dam, sesuai dengan masukkan pengguna. Prosesnya cukup efisien, mengingat jumlah node yang harus dilalui cukup banyak, namun tetap dapat menemukan semua kemunculan Dam dengan waktu yang relatif singkat. Memori yang dibutuhkan untuk melakukan proses ini relatif lebih lama dibandingkan dengan algoritma BFS, karena dilakukan secara rekursif.
Element: Dam, Algorithm: BFS, Count: 1

Hasil:

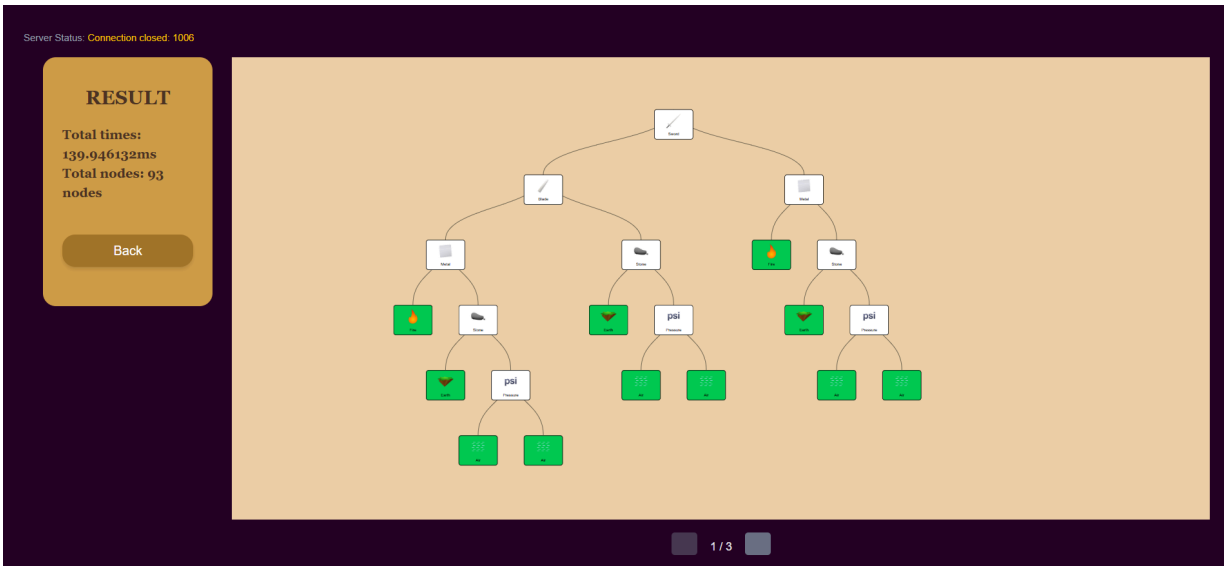


Analisis:

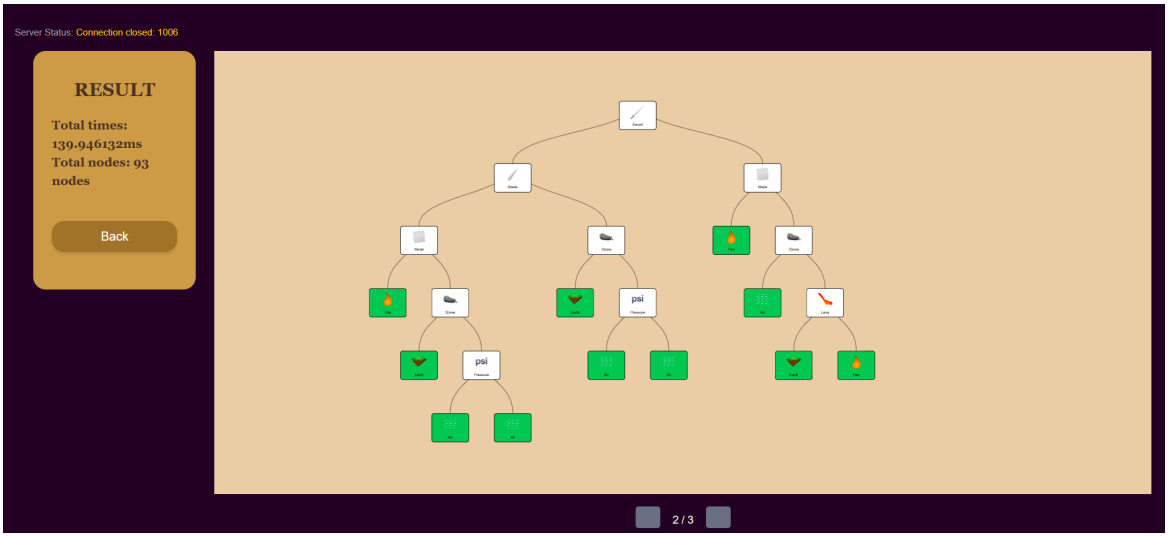
Dam berhasil ditemukan menggunakan algoritma BFS dalam waktu sekitar 211 milisecond dengan total 74 node yang telah dikunjungi selama proses pencarian berlangsung. Hasil yang didapat menampilkan 1 kombinasi resep yang memungkinkan untuk elemen Dam, sesuai dengan masukkan pengguna. Prosesnya cukup efisien, mengingat jumlah node yang harus dilalui cukup banyak, namun tetap dapat menemukan semua kemunculan Dam dengan waktu yang relatif singkat. Memori yang dibutuhkan untuk melakukan proses ini relatif lebih sedikit dibandingkan dengan algoritma DFS, namun membutuhkan waktu yang sedikit lebih lama.

Element: Sword, Algorithm: DFS, Count: 3

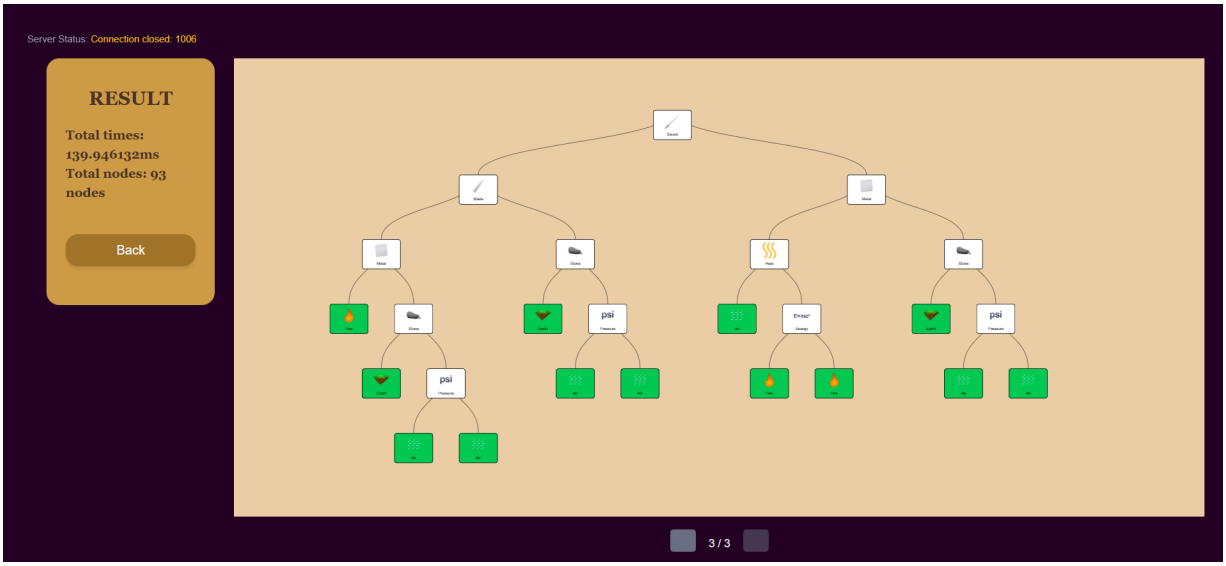
Hasil 1:



Hasil 2:



Hasil 3:

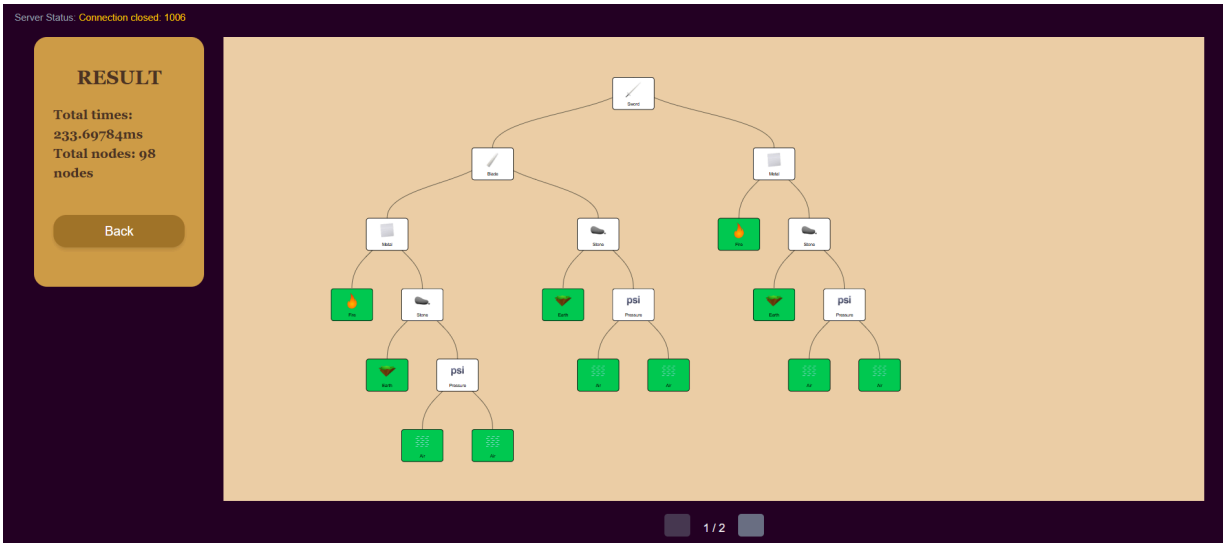


Analisis:

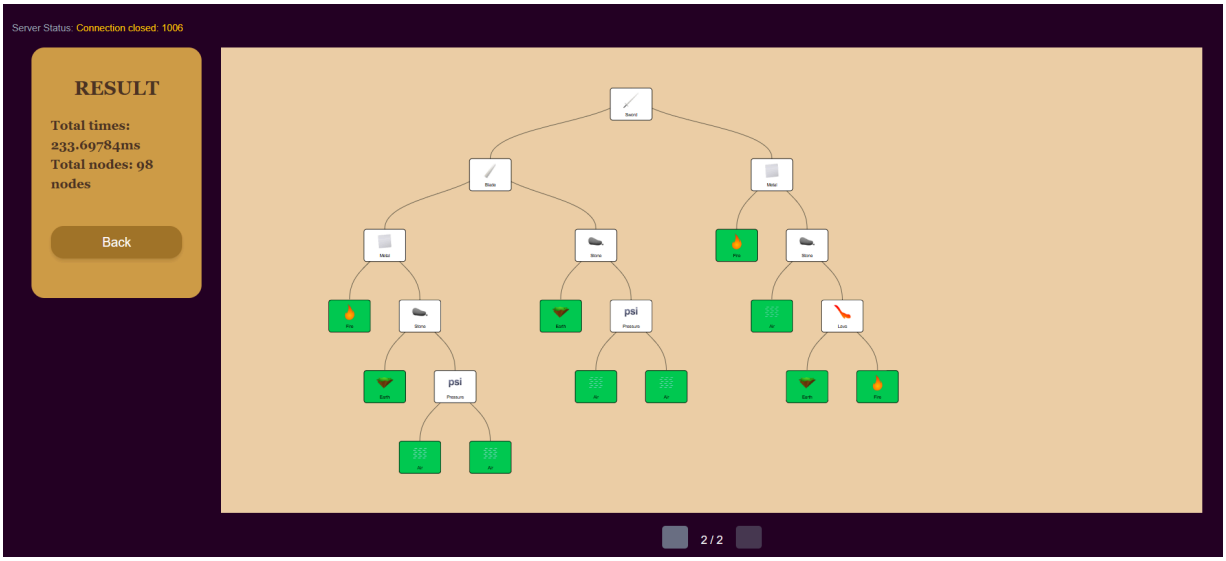
Sword berhasil ditemukan menggunakan algoritma DFS dalam waktu sekitar 0.138 detik dengan total 93 node yang telah dikunjungi selama proses pencarian berlangsung. Hasil yang didapat menampilkan 3 kombinasi resep yang memungkinkan untuk elemen sword, sesuai dengan masukkan pengguna. Prosesnya cukup efisien, mengingat jumlah node yang harus dilalui cukup banyak, namun tetap dapat menemukan semua kemunculan Sword dengan waktu yang relatif singkat. Memori yang dibutuhkan untuk melakukan proses ini relatif lebih lama dibandingkan dengan algoritma BFS, karena dilakukan secara rekursif.

Element: Sword, Algorithm: BFS, Count: 2

Hasil 1:



Hasil 2:

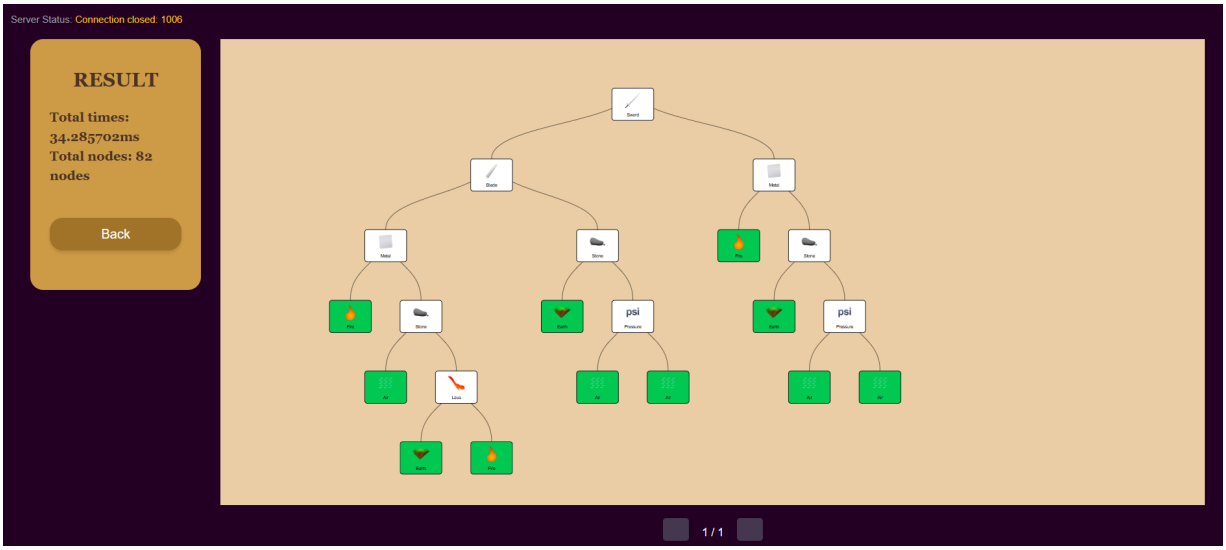


Analisis:

Sword berhasil ditemukan menggunakan algoritma BFS dalam waktu sekitar 0.233 detik dengan total 98 node yang telah dikunjungi selama proses pencarian berlangsung. Hasil yang didapat menampilkan 2 kombinasi resep yang memungkinkan untuk elemen sword, sesuai dengan masukkan pengguna. Prosesnya cukup efisien, mengingat jumlah node yang harus dilalui cukup banyak, namun tetap dapat menemukan semua kemunculan Sword dengan waktu yang relatif singkat. Memori yang dibutuhkan untuk melakukan proses ini relatif lebih sedikit dibandingkan dengan algoritma DFS, namun membutuhkan waktu yang sedikit lebih lama.

Element: Sword, Algorithm: Bidirectional, Count: 1

Hasil:

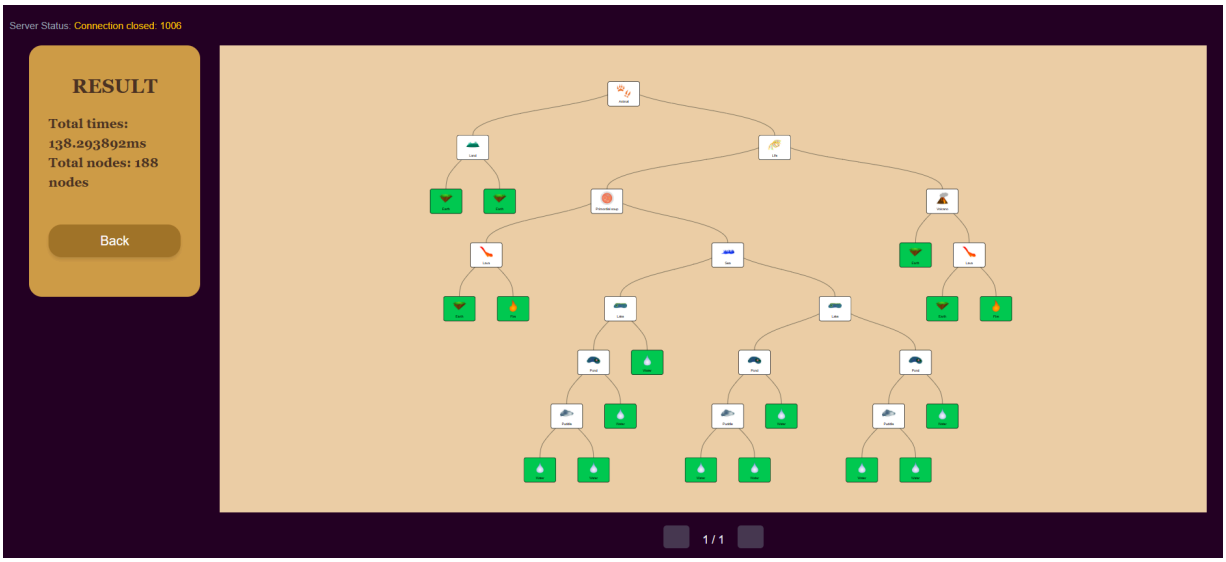


Analisis:

Sword berhasil ditemukan menggunakan algoritma Bidirectional dalam waktu sekitar 0.034 detik dengan total 82 node yang telah dikunjungi selama proses pencarian berlangsung. Hasil yang didapat menampilkan 1 kombinasi resep yang memungkinkan untuk elemen sword, sesuai dengan masukkan pengguna. Prosesnya cukup efisien, mengingat jumlah node yang harus dilalui cukup banyak, namun tetap dapat menemukan semua kemunculan Sword dengan waktu yang relatif singkat. Memori yang dan waktu yang dibutuhkan untuk melakukan proses ini relatif lebih sedikit dibandingkan BFS dan DFS, karena resep dicari berdasarkan dua arah yang saling menuju.

Element: Animal, Algorithm: DFS, Count: 1

Hasil:

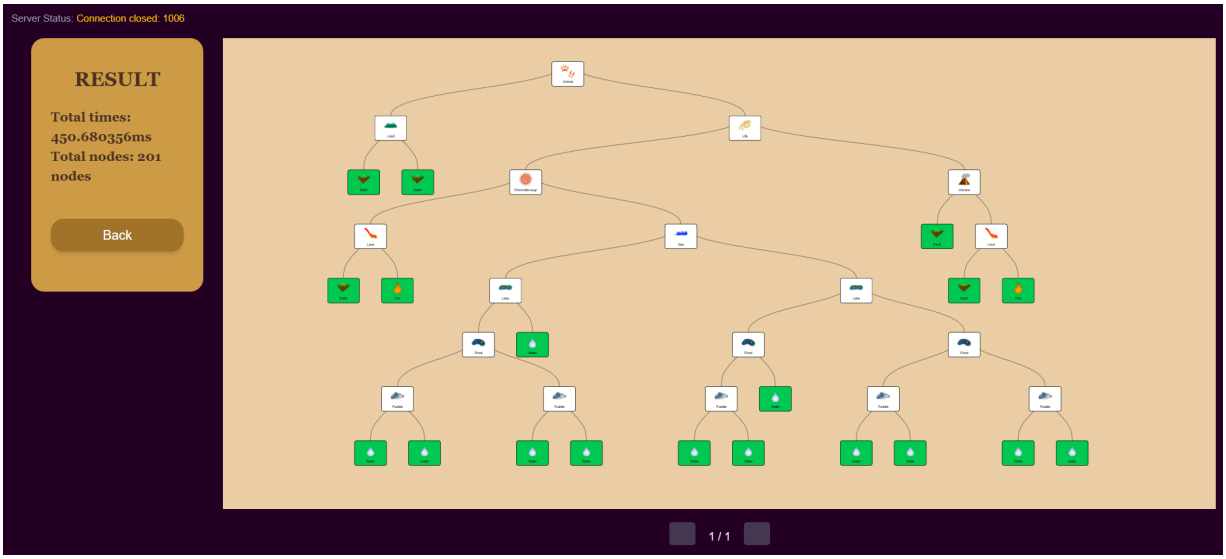


Analisis:

Animal berhasil ditemukan menggunakan algoritma DFS dalam waktu sekitar 138 milisecond dengan total 188 node yang telah dikunjungi selama proses pencarian berlangsung. Proses yang relatif lebih lama ini sesuai dengan tier animal yang berada pada level 9.

Element: Animal, Algorithm: BFS, Count: 1

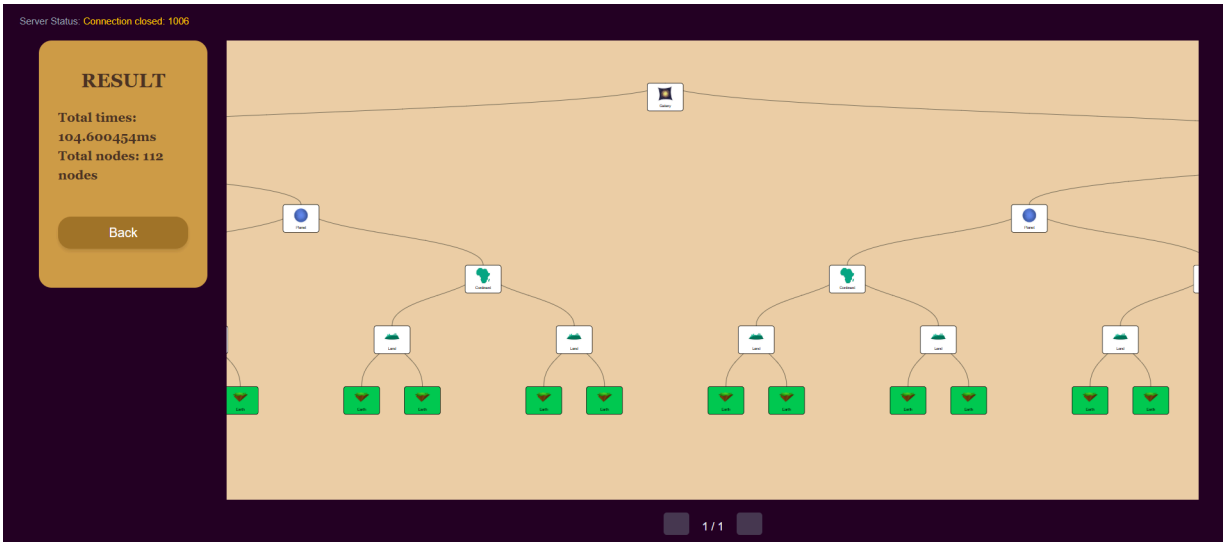
Hasil:



Analisis:
Animal berhasil ditemukan menggunakan algoritma BFS dalam waktu sekitar 450 milidetik, dengan total 201 simpul yang dikunjungi selama proses pencarian. Hasil ini konsisten dengan karakteristik BFS, yang mengunjungi semua simpul pada level yang sama terlebih dahulu sebelum menelusuri ke level berikutnya, berbeda dengan DFS yang langsung menelusuri simpul hingga ke kedalaman maksimal terlebih dahulu.

Element: Galaxy, Algorithm: DFS, Count: 1

Hasil:

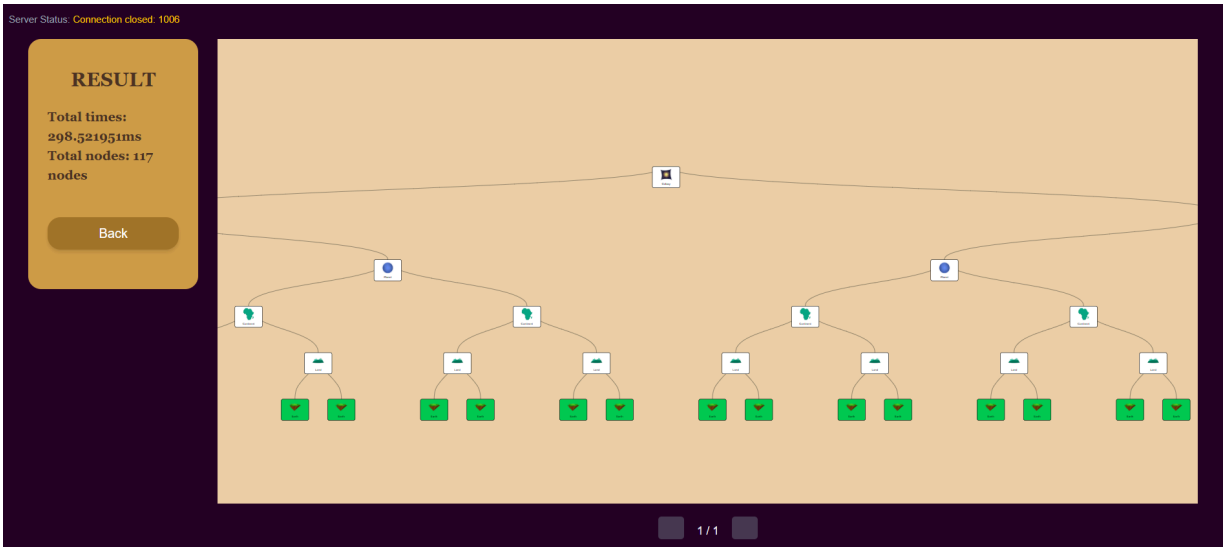


Analisis:

Galaxy berhasil ditemukan menggunakan algoritma DFS dalam waktu sekitar 104 milisecond dengan total 112 node yang telah dikunjungi selama proses pencarian berlangsung. Hasil yang didapat menampilkan 1 kombinasi resep yang memungkinkan untuk elemen Galaxy, sesuai dengan masukkan pengguna. Prosesnya cukup efisien, mengingat jumlah node yang harus dilalui cukup banyak, namun tetap dapat menemukan semua kemunculan Galaxy dengan waktu yang relatif singkat. Memori yang dibutuhkan untuk melakukan proses ini relatif lebih lama dibandingkan dengan algoritma BFS, karena dilakukan secara rekursif.

Element: Galaxy, Algorithm: BFS, Count: 1

Hasil:



Analisis:

Dam berhasil ditemukan menggunakan algoritma BFS dalam waktu sekitar 298 milisecond dengan total 117 node yang telah dikunjungi selama proses pencarian berlangsung. Hasil yang didapat menampilkan 1 kombinasi resep yang memungkinkan untuk elemen Galaxy, sesuai dengan masukkan pengguna. Prosesnya cukup efisien, mengingat jumlah node yang harus dilalui cukup banyak, namun tetap dapat menemukan semua kemunculan Galaxy dengan waktu yang relatif singkat. Memori yang dibutuhkan untuk melakukan proses ini relatif lebih sedikit dibandingkan dengan algoritma DFS, namun membutuhkan waktu yang sedikit lebih lama.

V. PENUTUP

5.1. Kesimpulan

Dalam Tugas Besar 2 IF2211 Strategi Algoritma, kami telah mengembangkan aplikasi pencarian resep pembuatan elemen untuk permainan Little Alchemy 2 menggunakan algoritma BFS dan DFS serta *Bidirectional Search* dalam bahasa Go. Dengan mengimplementasikan algoritma tersebut dengan baik, dan memanfaatkan fitur *multithreading* pada bahasa Go, kami berhasil membuat aplikasi bekerja dengan lancar dan cepat, bahkan untuk jumlah resep yang cukup besar. Setelah mencoba aplikasi yang telah kami buat secara langsung dengan permainan Little Alchemy 2, aplikasi berhasil mendapatkan resep untuk elemen-elemen dengan benar

5.2. Saran

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

1. Lebih mengoptimalkan manajemen waktu dalam pengerjaan Tugas Besar agar dapat lebih menyempurnakan dan mengoptimalkan program.
2. Membaca dan memahami dokumentasi dan tata cara penggunaan bahasa pemrograman yang digunakan secara lengkap agar pekerjaan tugas menjadi lebih terstruktur dan lancar.
3. Melakukan testing secara berulang-ulang hingga tidak ditemukan kesalahan.

5.3. Refleksi

Pengerjaan tugas besar ini menjadi pengalaman yang cukup berkesan karena adanya teknik-teknik pemrograman baru seperti *multithreading* dan bahasa Go yang kami kurang familiar dengannya. Meskipun awalnya terasa menantang dan asing, proses pembelajaran dan implementasi membuka wawasan kami. Tugas besar ini memperkaya pemahaman kami mengenai materi-materi yang diajarkan di kelas, seperti algoritma BFS, DFS, dan algoritma yang tidak diajarkan di kelas yaitu *Bidirectional Search*. Selain itu, kami juga mendapat pemahaman baru terkait cara penggunaan bahasa Go dan websocket. Oleh karena itu, hal ini memungkinkan kami untuk memahami materi lebih dalam karena kami dapat melihat penerapannya dalam kehidupan nyata.

LAMPIRAN

Tautan Repository Github

Berikut tautan repository frontend github kelompok kami untuk Tugas Besar 2 IF2211 Strategi Algoritma :
https://github.com/azfaradhi/Tubes2_FE_Bolang

Berikut tautan repository backend github kelompok kami untuk Tugas Besar 2 IF2211 Strategi Algoritma :
https://github.com/Ferdin-Arsenic/Tubes2_BE_Bolang

Tautan Video

Berikut tautan video kelompok kami untuk Tugas Besar 2 IF2211 Strategi Algoritma :
Demo Tugas Besar 2 IF2211-Strategi Algoritma || Algoritma BFS dan DFS dalam Game Little Alchemy 2

Tabel Kelengkapan Spesifikasi

No	Poin	Ya	Tidak
1	Aplikasi dapat dijalankan.	✓	
2	Aplikasi dapat memperoleh data *recipe* melalui scraping.	✓	
3	Algoritma *Depth First Search* dan *Breadth First Search* dapat menemukan *recipe* elemen dengan benar.	✓	
4	Aplikasi dapat menampilkan visualisasi *recipe* 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 *Bidirectional*.	✓	
9	Membuat bonus *Live Update*.	✓	

10	Aplikasi di-*containerize* dengan Docker.	✓	
11	Aplikasi di-*deploy* dan dapat diakses melalui internet.	✓	

DAFTAR PUSTAKA

Munir, Rinaldi. 2025. *Breadth First Search (BFS) dan Depth First Search (DFS) - Bagian 1*. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf (Diakses pada 12 Mei 2025).

Munir, Rinaldi. 2025. *Breadth First Search (BFS) dan Depth First Search (DFS) - Bagian 2*. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-(2025)-Bagian2.pdf (Diakses pada 12 Mei 2025).

Kontributor Fandom. 2025. Elements (Little Alchemy 2) - Little Alchemy Wiki. https://little-alchemy.fandom.com/wiki/Elements_(Little_Alchemy_2 (Diakses pada 12 Mei 2025).

Penulis Dokumentasi Go. 2025. Documentation - The Go Programming Language. https://go.dev/doc/ (Diakses pada 12 Mei 2025).