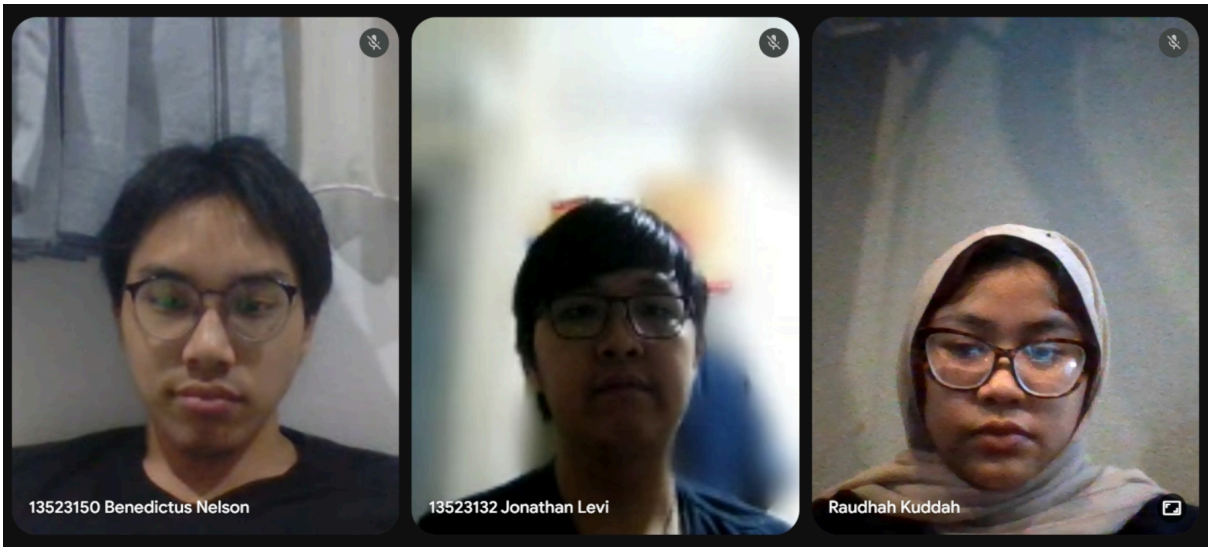
Pemanfaatan Algoritma Breadth First Search dan Depth First Search Dalam Pencarian Elemen Pada Permainan Little Alchemy 2

Laporan Tugas Besar 2

IF2211 Strategi Algoritma



Disusun Oleh Kelompok 16 (Fullmetal JavaScript):

13122003	Raudhah Yahya Kuddah
13523132	Jonathan Levi
13523150	Benedictus Nelson

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2025

DAFTAR ISI

DAFTAR ISI **2**
Bab 1
Deskripsi Tugas **3**
Bab 2
Landasan Teori **5**
2.1 Dasar Teori 5
2.2 Penjelasan Singkat 6
Bab 3
Analisis Pemecahan Masalah **8**
3.1 Langkah-langkah pemecahan masalah 8
3.2 Proses pemetaan masalah menjadi elemen-elemen 9
3.3 Fitur fungsional dan arsitektur aplikasi web yang dibangun 11
3.4 Contoh ilustrasi kasus 13
Bab 4
Implementasi dan Pengujian **15**
4.1 Spesifikasi Teknis Program 15
4.2 Tata Cara Penggunaan Program 16
4.3 Hasil Pengujian 17
4.4 Analisis Hasil Pengujian 19
Bab 5
Kesimpulan dan Saran **20**
a. Kesimpulan 20
b. Saran 20
LAMPIRAN **22**
DAFTAR PUSTAKA **23**

Bab 1

Deskripsi Tugas

Gambar 1. Little Alchemy 2
(sumber: https://www.thegamer.com)

Little Alchemy 2 merupakan permainan berbasis web dan aplikasi yang menantang pemain untuk menciptakan berbagai macam elemen turunan dengan cara menggabungkan empat elemen dasar, yaitu air, api, tanah, dan udara. Permainan ini dikembangkan oleh Recloak dan dirilis pada tahun 2017 sebagai sekuel dari Little Alchemy yang diluncurkan pada tahun 2010. Tujuan utama dari permainan ini adalah untuk menemukan seluruh 720 elemen dari 4 elemen dasar yang tersedia (*air, earth, fire,* dan *water*) melalui kombinasi-kombinasi elemen yang valid.

Pada tugas besar Strategi Algoritma ini, mahasiswa diminta untuk memodelkan dan menyelesaikan permainan ***Little Alchemy 2*** menggunakan dua pendekatan pencarian, yaitu ***Depth First Search*** (DFS) dan ***Breadth First Search*** (BFS). Kedua strategi pencarian ini digunakan untuk menelusuri kemungkinan kombinasi elemen dalam menemukan jalur pembentukan elemen-elemen turunan tertentu secara sistematis.

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.

Kompleksitas permainan ini terletak pada banyaknya jumlah kombinasi yang dapat dilakukan serta fakta bahwa elemen-elemen turunan dapat digunakan kembali untuk membentuk elemen lain, sehingga menciptakan pohon kombinasi yang sangat luas dan dalam. Selain itu, tidak semua kombinasi menghasilkan elemen baru, sehingga strategi pencarian yang efisien sangat diperlukan untuk menghindari eksplorasi yang sia-sia.

Laporan ini akan membahas bagaimana permainan ***Little Alchemy 2*** dapat direpresentasikan dalam bentuk graf eksplisit, serta bagaimana penerapan DFS dan BFS dapat diimplementasikan untuk menelusuri kemungkinan kombinasi elemen. Untuk mendapatkan elemen turunan pemain dapat melakukan *combine* antara 2 elemen untuk menghasilkan elemen baru. Elemen turunan yang telah didapatkan dapat digunakan kembali oleh pemain untuk membentuk elemen lainnya.

Bab 2
Landasan Teori

2.1 Dasar Teori

2.1.1 Penjelajahan Graf

Graf merupakan representasi struktur data yang terdiri dari sekumpulan simpul (*nodes*) dan relasi antar simpul tersebut yang disebut sisi (*edges*). Dalam pengaplikasiannya, graf dapat berupa berarah (*directed*) maupun tidak berarah (*undirected*), tergantung pada sifat relasi yang dimodelkan.

Pada permainan seperti *Little Alchemy 2*, setiap elemen dalam permainan dapat dipetakan sebagai simpul dalam graf, sedangkan hubungan antar elemen yang memungkinkan terbentuknya elemen baru direpresentasikan sebagai sisi. Pada permainan *Little Alchemy 2* graf berupa berarah (*directed*).

Proses menjelajahi atau menelusuri graf bertujuan untuk mengunjungi simpul-simpul tertentu, baik untuk menemukan jalur ke simpul tujuan, mengeksplorasi seluruh graf, atau mencari solusi tertentu. Dua pendekatan dasar yang banyak digunakan dalam proses ini adalah algoritma *Breadth First Search* (BFS) dan *Depth First Search* (DFS). Kedua algoritma ini memiliki perbedaan dalam cara mereka menjelajah graf dan masing-masing memiliki kelebihan serta keterbatasan tersendiri.

2.1.2 Algoritma Breadth First Search (BFS)

Algoritma BFS bekerja dengan menjelajahi graf berdasarkan tingkat kedalaman simpul secara bertahap. Dimulai dari simpul awal, BFS akan mengunjungi semua simpul yang terhubung langsung sebelum melanjutkan ke simpul-simpul pada level berikutnya. Untuk menjaga urutan ini, BFS memanfaatkan struktur data *queue* (antrian), di mana simpul yang pertama masuk akan dieksekusi lebih dahulu.

Dalam konteks permainan *Little Alchemy 2*, BFS cocok untuk menemukan kombinasi elemen dengan jumlah langkah minimum, karena sifat traversal-nya yang sistematis dan menyebar secara luas. Namun demikian, BFS bisa menjadi tidak efisien dari sisi memori jika graf yang dibentuk memiliki tingkat percabangan yang tinggi, karena banyaknya simpul yang harus disimpan pada tiap level eksplorasi.

2.1.3 Algoritma Depth First Search (DFS)

Berbeda dengan BFS, algoritma DFS memprioritaskan eksplorasi jalur sampai ke simpul terdalam terlebih dahulu sebelum kembali (*backtrack*) untuk mencoba jalur lain. DFS biasanya menggunakan stack baik eksplisit maupun melalui rekursi dalam implementasinya.

Keunggulan DFS adalah efisiensi penggunaan memorinya, karena tidak perlu menyimpan semua simpul pada level yang sama. DFS juga berguna untuk menjelajahi semua kemungkinan jalur yang ada. Namun, DFS tidak menjamin solusi paling pendek dan bisa saja mengeksplorasi jalur yang tidak relevan terlalu dalam sebelum mencapai simpul target.

2.2 Penjelasan Singkat

Aplikasi web yang dikembangkan dalam tugas ini berfungsi sebagai alat bantu visualisasi dan pencarian jalur pembuatan elemen dalam permainan Little Alchemy 2 berdasarkan pendekatan graf. Aplikasi ini tidak mereplikasi keseluruhan permainan, melainkan menekankan pada aspek algoritmis, yakni bagaimana sebuah elemen dapat dibentuk dari kombinasi elemen-elemen lain melalui representasi struktur graf.

Setiap elemen dalam permainan diperlakukan sebagai simpul dalam graf, sementara kombinasi atau recipe antar elemen direpresentasikan sebagai sisi. Ketika pengguna memasukkan nama elemen yang ingin dicari, aplikasi akan membangun graf dari elemen dasar hingga mencapai target, lalu menggunakan algoritma *Breadth First Search* (BFS) atau *Depth First Search* (DFS) untuk menelusuri jalur-jalur kemungkinan pembentukan elemen tersebut.

Tujuan utama dari aplikasi ini adalah untuk menunjukkan urutan kombinasi (*recipe path*) dari elemen dasar hingga ke elemen target, bukan untuk melakukan simulasi atau drag-and-drop

seperti pada permainan aslinya. Hasil pencarian ditampilkan secara terstruktur dalam antarmuka web, sehingga pengguna dapat memahami langkah-langkah pembentukan elemen secara logis berdasarkan struktur data graf yang dibentuk.

Aplikasi ini dirancang untuk mendemonstrasikan penerapan konsep algoritma pencarian graf secara praktis dalam konteks permainan populer, sekaligus membantu pengguna memahami kompleksitas di balik pembentukan elemen yang tampaknya sederhana.

Bab 3
Analisis Pemecahan Masalah

3.1 Langkah-langkah pemecahan masalah.

Penyelesaian masalah dalam pengembangan aplikasi ini dilakukan secara bertahap dan sistematis, dimulai dari proses pengumpulan data hingga penyajian visual di sisi pengguna. Berikut adalah alur langkah-langkah yang ditempuh:

3.1.1 Pengumpulan dan Pengolahan Data

Langkah pertama dimulai dengan *web scraping* dari situs wiki resmi *Little Alchemy 2* yang menyimpan semua data kombinasi elemen. Data yang diperoleh berupa daftar elemen beserta pasangan pembentuknya (*recipes*). Hasil scraping kemudian diolah dan disusun ke dalam struktur data relasional yang mudah dikonversi ke bentuk graf. Seluruh informasi disimpan dalam format dictionary atau peta, dengan elemen sebagai kunci dan list pasangan penyusunnya sebagai nilai.

3.1.2 Pembangunan Struktur Graf di Backend

Dari data hasil preprocessing, dibangun struktur graf berarah (*directed graph*) yang merepresentasikan hubungan antar elemen. Setiap simpul (*node*) adalah elemen, dan sisi (*edge*) menunjukkan jalur kombinasi. Graf ini disimpan sebagai struktur adjacency list, yang memungkinkan efisiensi dalam penelusuran simpul tetangga selama algoritma berjalan.

3.1.3 Implementasi Algoritma Pencarian (BFS & DFS)

Setelah struktur graf terbangun, dilanjutkan ke penerapan algoritma pencarian untuk menemukan jalur kombinasi dari elemen target. BFS bekerja dengan menjelajahi graf secara melebar. BFS akan menjelajahi semua kemungkinan kombinasi di tingkat awal dahulu sebelum ke langkah kombinasi berikutnya, strategi ini menjamin langkah terpendek menuju elemen target. BFS menggunakan struktur *queue* serta melacak elemen yang telah dikunjungi agar tak terulang. Sedangkan DFS menggunakan pendekatan yang sebaliknya yaitu menjelajahi secara mendalam sebelum kembali dan mencoba jalur lain. Algoritma DFS memiliki kompleksitas penyimpanan lebih rendah dibandingkan BFS pada kondisi tertentu. Kedua algoritma tidak hanya mengembalikan

jalur tunggal, namun multiple atau banyak sesuai jumlah yang diminta pengguna. Ini menggunakan multithreading agar lebih efisien.

3.1.4 Pembangunan Struktur Pohon (*Recipe Tree*)

Hasil dari proses pencarian diubah menjadi struktur tree untuk memudahkan visualisasi. Pohon dibangun secara rekursif dari elemen target hingga ke elemen-elemen dasar, dengan setiap node menyimpan nama elemen, urutan kombinasi, dan ID unik. Struktur ini memastikan kejelasan dalam menyusun recipe bertingkat.

3.1.5 Pembuatan REST API

Setelah struktur recipe tree terbentuk, data disiapkan dalam format JSON dan disediakan ke sisi frontend melalui RESTful API. API ini menerima permintaan elemen target dan algoritma yang dipilih baik BFS atau DFS, lalu mengembalikan hasil pencarian berupa struktur pohon kombinasi dalam format JSON yang telah dirapikan.

3.1.6 Visualisasi di Frontend

Di sisi frontend, JSON yang dikirimkan oleh backend diolah untuk ditampilkan secara interaktif. Node ditampilkan dalam bentuk pohon hierarkis, dengan jalur kombinasi menuju kepada elemen target yang terdapat di bagian bawah. Pengguna dapat memilih algoritma BFS atau DFS, mengatur jumlah recipe yang ingin ditampilkan, dan menelusuri struktur pohon secara interaktif.

3.2 Proses pemetaan masalah menjadi elemen-elemen

Permainan *Little Alchemy 2* memiliki prinsip dasar bahwa setiap elemen kompleks terbentuk dari kombinasi dua elemen lain yang lebih sederhana. Kombinasi ini disebut sebagai *recipe* atau resep. Setiap *recipe* direpresentasikan sebagai himpunan dua elemen yang membentuk satu elemen hasil. Berdasarkan konsep ini, sistem pencarian dan visualisasi resep dikembangkan dengan struktur data graf dan pohon.

Struktur data yang digunakan dalam *backend* dijelaskan dalam berkas models.go, yang terdiri dari:

- TreeNode : mewakili node dalam pohon komposisi elemen. Setiap node menyimpan nama elemen dan juga *children* dari node tersebut yang merupakan elemen penyusun.

- TreeResult : Struktur hasil pencarian yang dikirim ke *frontend.* Berisikan node root dari pohon, nama algoritma pencarian yang digunakan, durasi pencarian, dan jumlah node yang dikunjungi.
- SearchRequest : Struktur permintaan pencarian elemen, mencakup nama elemen target, algoritma yang dipilih baik itu BFS atau DFS, serta batas jumlah hasil yang ingin dikembalikan.

3.2.1 Scraping Data

File scrapper.go bertugas mengambil data langsung dari halaman wiki Little Alchemy 2. Fungsi ScrapeElement(elementName) melakukan scraping terhadap halaman elemen tertentu untuk mengekstrak semua pasangan bahan (dalam format [][]string) di bawah heading "Recipes". Elemen dasar seperti air, bumi, api, dan udara ditandai secara statis agar tidak dicari lebih lanjut dan menjadi ujung pohon (daun).

3.2.2 Graph Search

File search.go menangani proses pencarian elemen menggunakan algoritma BFS maupun DFS. Dari elemen target, pencarian dilakukan mundur ke elemen dasar. Hasilnya adalah node root pohon (TreeNode), lengkap dengan informasi durasi pencarian dan jumlah node yang dikunjungi. Fungsi buildRecipeTree bekerja secara rekursif membentuk struktur pohon dari elemen target ke elemen dasar.

3.2.3 HTTP API (Handler)

File handler.go berperan sebagai penghubung antara frontend dan backend. Saat pengguna mengirim permintaan pencarian, handler akan:

- Men-*decode* SearchRequest dari body POST
- Panggil search.Search(req)
- Pangkas pohon jika terlalu besar (pruneTreeNodes)
- Men-*encode* TreeResult jadi JSON *response*

3.2.4 *Frontend* (Next.js dan React)

Bagian frontend menggunakan Next.js dan React sebagai kerangka kerja utama. Berkas index.js menjadi halaman utama tempat pengguna mengisi formulir pencarian. Komponen SearchForm.js mengatur logika pengiriman request ke endpoint /search.

Visualisasi pohon dilakukan melalui dua pendekatan:

- RecipeVisualizer.js menggunakan pustaka react-d3-tree untuk menampilkan pohon secara vertikal.
- RecipeTree.js menggunakan React Flow untuk menyusun node dan edge secara fleksibel dan interaktif

3.2.5 Styling dan Visualisasi

Tampilan antarmuka disempurnakan dengan penyesuaian CSS pada global.css agar visualisasi terlihat rapi dan informatif. Layout dan warna didesain agar nyaman dilihat dan intuitif bagi pengguna awam.

3.3 Fitur fungsional dan arsitektur aplikasi web yang dibangun.

Fitur fungsional dari aplikasi ini meliputi:

1. Pencarian Resep
   - Pengguna memasukkan nama elemen (misal 'brick') dan memilih algoritma (BFS/DFS) serta jumlah hasil.
   - *Frontend* (SearchForm.js) mengirim POST /search dengan body JSON models.SearchRequest.
2. *Scraping* Data
   - *Backend* mengambil resep dari wiki Little Alchemy 2.
   - Memetakan element ke daftar pasangan bahan ([][]string).
3. Bangun Pohon Resep
   - Fungsi buildRecipeTree di search/search.go merekursi dari target ke elemen dasar (air, bumi, api, air).
   - Hasilnya *models.TreeNode yang berisi Name dan Children.
4. Algoritma BFS & DFS

- Kedua fungsi (BFS, DFS) di search/search.go menghitung durasi, jumlah node terkunjungi, lalu membungkusnya ke models.TreeResult.

5. HTTP API

   - *Handler* /search di handler.go menerima permintaan, memanggil search.Search, memangkas pohon jika terlalu besar, lalu mengembalikan JSON.

6. Visualisasi Pohon

   Frontend menggunakan React:

   - RecipeTree.js dengan React Flow untuk meng-generate node/edge dan merender tree secara grafis.
   - RecipeVisualizer.js menggunakan react-d3-tree.

7. Penggunaan Lingkungan & Deploy

   Frontend mengandalkan NEXT_PUBLIC_API_URL untuk URL API.

   - Dockerfile (*backend*): *image* Go statis.
   - Dockerfile (*frontend*): *build* dan *serve* Next.js.
   - docker-compose.yml: menyatukan keduanya dalam satu network.

Arsitektur aplikasi

Aplikasi pencari resep *Little Alchemy 2* ini dibangun dengan menggunakan arsitektur web yang terbagi menjadi dua bagian utama yaitu *frontend* dan *backend*. Selain itu, aplikasi juga memiliki bagian khusus untuk pengambilan data dari internet yaitu *web scraper* dan penerapan algoritma BFS dan DFS.

Pada bagian *frontend,* pengguna akan berinteraksi langsung melalui antarmuka berbasis website yang dibangun menggunakan Next.js dan React. Di halaman utama, pengguna dapat mengetik nama elemen yang ingin dicari (misalnya “brick”), lalu memilih jenis algoritma pencarian yang ingin digunakan (BFS atau DFS). Setelah itu, aplikasi akan mengirimkan permintaan pencarian ke server *backend* melalui sebuah API *endpoint* menggunakan metode HTTP POST. Permintaan ini diterima oleh bagian *backend*, yang merupakan bagian dari program yang berjalan di server. Backend akan membaca data dari permintaan tersebut, lalu memprosesnya. Untuk melakukan pencarian, *backend* membutuhkan data semua resep kombinasi elemen yang ada di permainan. Di sinilah peran scraper digunakan. Scraper akan

secara otomatis membuka halaman wiki *Little Alchemy 2* dan mengambil semua data kombinasi elemen dari sana, seperti "*fire* + *water* = *steam*".

Dengan data tersebut, backend kemudian membangun sebuah struktur graf, yaitu kumpulan elemen yang terhubung berdasarkan kombinasi pembentukan elemen. Algoritma pencarian seperti *Breadth-First Search* (BFS) atau *Depth-First Search* (DFS) kemudian digunakan untuk mencari jalur kombinasi dari elemen-elemen dasar (seperti air, api, tanah, udara) hingga ke elemen yang diinginkan oleh pengguna. Hasil pencarian ini dikonversi menjadi bentuk pohon (*tree*), di mana setiap simpul (*node*) mewakili satu elemen, dan cabangnya menunjukkan dari mana elemen tersebut berasal. Struktur pohon ini kemudian dikirim kembali ke *frontend* dalam format JSON, sebuah format data ringan yang mudah dibaca oleh JavaScript.

Setelah menerima data, *frontend* menampilkan hasil pencarian dalam bentuk visual pohon interaktif menggunakan pustaka *React Flow*. Pengguna bisa melihat dengan jelas bagaimana sebuah elemen terbentuk dari gabungan elemen-elemen lainnya, secara bertingkat dari atas ke bawah. Dengan arsitektur seperti ini, sistem menjadi modular dan terpisah dengan jelas: frontend menangani tampilan dan interaksi pengguna, *backend* menangani logika pencarian, scraper mengambil data dari sumber eksternal, dan API menjadi penghubung antar komponen. Pendekatan ini mempermudah pengembangan, pemeliharaan, dan perluasan fitur di masa mendatang.

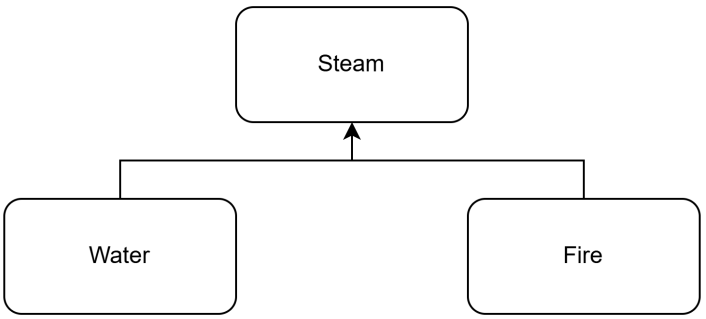
3.4 Contoh ilustrasi kasus.

Sebagai contoh, misalkan pengguna memasukkan kata kunci "Steam" untuk melakukan pencarian resep elemen. Sistem kemudian akan menelusuri bagaimana elemen Steam dapat dibuat, dan dari mana bahan-bahannya berasal. Dalam hal ini, Steam diketahui dapat dibuat dari kombinasi dua elemen dasar, yaitu Water dan Fire. Karena Water dan Fire merupakan elemen dasar yang tidak berasal dari kombinasi elemen lain, maka penelusuran berhenti di sana.

Struktur hasil pencarian ini divisualisasikan dalam bentuk pohon (tree), di mana setiap simpul (node) mewakili satu elemen, dan cabangnya menunjukkan elemen-elemen yang digunakan untuk membuatnya. Untuk elemen "Steam", pohon hasil penelusurannya dapat digambarkan sebagai berikut:

```
name: "Steam",
  children: [
    {
      name: "Water + Fire",
      children: [
        {
          name: "Water",
          children: []
        },
        {
          name: "Fire",
          children: []
        }
      ]
    }
  ]
```

Visualisasi seperti ini memudahkan pengguna dalam memahami bagaimana suatu elemen terbentuk, serta melihat dengan jelas elemen-elemen penyusunnya, tanpa perlu membongkar elemen dasar yang sudah diketahui.



Bab 4
Implementasi dan Pengujian

4.1 Spesifikasi Teknis Program

4.1.1 Handler

Script ini berfungsi sebagai handler HTTP dalam backend aplikasi untuk menangani permintaan pencarian melalui *endpoint* /search. Secara garis besar, fungsi utama dari script ini adalah untuk menerima permintaan pencarian dari frontend, memrosesnya, dan mengembalikan hasil pencarian dalam format JSON. Proses ini dilakukan dengan menggunakan dua algoritma pencarian, yaitu *Breadth-First Search* (BFS) dan *Depth-First Search* (DFS), tergantung pada preferensi yang diberikan dalam permintaan. Apabila tidak ada algoritma yang ditentukan, maka algoritma default yang digunakan adalah BFS.

Setelah data permintaan diproses, dilakukan pemanggilan fungsi search.Search untuk menjalankan pencarian sesuai dengan elemen dan algoritma yang ditentukan. Hasil pencarian kemudian diproses untuk memangkas jumlah node yang ditampilkan, dengan membatasi jumlah maksimal node yang akan dikembalikan, yaitu 1000. Hal ini dilakukan dengan menggunakan fungsi pruneTreeNodes, yang akan memangkas cabang pohon komposisi elemen agar tidak melebihi jumlah node yang ditentukan.

Terakhir, hasil pencarian yang telah diproses akan dikembalikan ke frontend dalam bentuk JSON. Jika terdapat kesalahan selama proses pengolahan atau pengiriman hasil, server akan mengirimkan respons error yang sesuai. Fungsi RegisterRoutes bertugas untuk mendaftarkan handler ini ke dalam multiplexer HTTP agar dapat diakses melalui *endpoint* yang telah ditentukan.

4.1.2 Scrapper

Script ini berfungsi sebagai alat untuk melakukan web *scraping* pada halaman elemen di website *Little Alchemy* dan mengumpulkan data resep-resep yang terdiri dari dua elemen pembentuk. Fungsi utama dari script ini adalah untuk mengambil nama elemen, membangun URL yang sesuai, dan mengekstrak informasi resep-resep yang dapat dibentuk dari elemen tersebut, kemudian menyimpannya dalam struktur

data yang terorganisir. Data hasil *scraping* ini kemudian dapat digunakan untuk membangun pohon komposisi elemen, seperti yang diharapkan dalam konteks aplikasi ini.

Fungsi utama dalam script ini adalah ScrapeElement, yang bertanggung jawab untuk mengambil data dari halaman web berdasarkan nama elemen yang diberikan. Fungsi ini pertama-tama membangun URL untuk halaman elemen dengan mengganti spasi dalam nama elemen menjadi garis bawah. Kemudian, fungsi ini melakukan HTTP GET request ke URL tersebut dan memeriksa status kode respons dari server. Jika respons berhasil (status 200), fungsi akan melanjutkan untuk mem-parsing isi halaman HTML menggunakan library GoQuery. Proses parsing dilakukan untuk mencari bagian yang berjudul "Recipes" pada halaman, di mana resep-resep elemen disajikan. Setelah menemukan bagian tersebut, setiap item daftar <li> yang berisi dua elemen pembentuk akan diproses dan disimpan dalam array *recipes*.

Setelah selesai mengekstrak resep-resep yang valid, fungsi ini akan mengembalikan hasil dalam bentuk struct ElementData, yang berisi daftar pasangan elemen yang dapat membentuk elemen yang dicari. Jika terjadi kesalahan selama proses scraping atau parsing, fungsi ini akan mengembalikan error yang sesuai. Dengan demikian, script ini memungkinkan pengguna untuk melakukan scraping informasi elemen secara otomatis dan terstruktur dari halaman *Little Alchemy*.

4.1.3 Search

Script ini menyediakan dua algoritma pencarian, yaitu *Breadth-First Search* (BFS) dan *Depth-First Search* (DFS), untuk membangun pohon resep elemen berdasarkan input dari pengguna. Fungsi utama Search akan menentukan algoritma pencarian yang digunakan berdasarkan parameter dalam permintaan (*request*) dan kemudian memanggil fungsi BFS atau DFS sesuai dengan algoritma yang dipilih. Kedua algoritma ini bekerja dengan membangun pohon resep secara rekursif melalui fungsi buildRecipeTree, yang akan membangun struktur pohon dari elemen yang dicari beserta resep-resep yang dapat dibuat dari elemen-elemen tersebut. Proses ini juga mempertimbangkan elemen dasar, yang tidak memiliki resep lebih lanjut, dan

mencegah siklus dengan mencatat elemen yang sudah dikunjungi menggunakan *map visited.*

Pohon resep dibangun dengan cara iteratif, di mana setiap elemen akan diperiksa apakah merupakan elemen dasar atau memiliki resep yang dapat membentuknya. Setiap resep yang ditemukan akan dikombinasikan dengan elemen lainnya untuk membentuk pohon yang lebih besar, yang mencakup elemen-elemen yang lebih kompleks. Selain itu, waktu eksekusi pencarian dan jumlah node yang dikunjungi dicatat untuk setiap pencarian dan dikembalikan dalam objek TreeResult. Hasil pencarian ini memberikan gambaran lengkap mengenai bagaimana elemen yang dicari dapat dibentuk melalui kombinasi elemen-elemen lainnya dengan menggunakan algoritma pencarian BFS atau DFS.

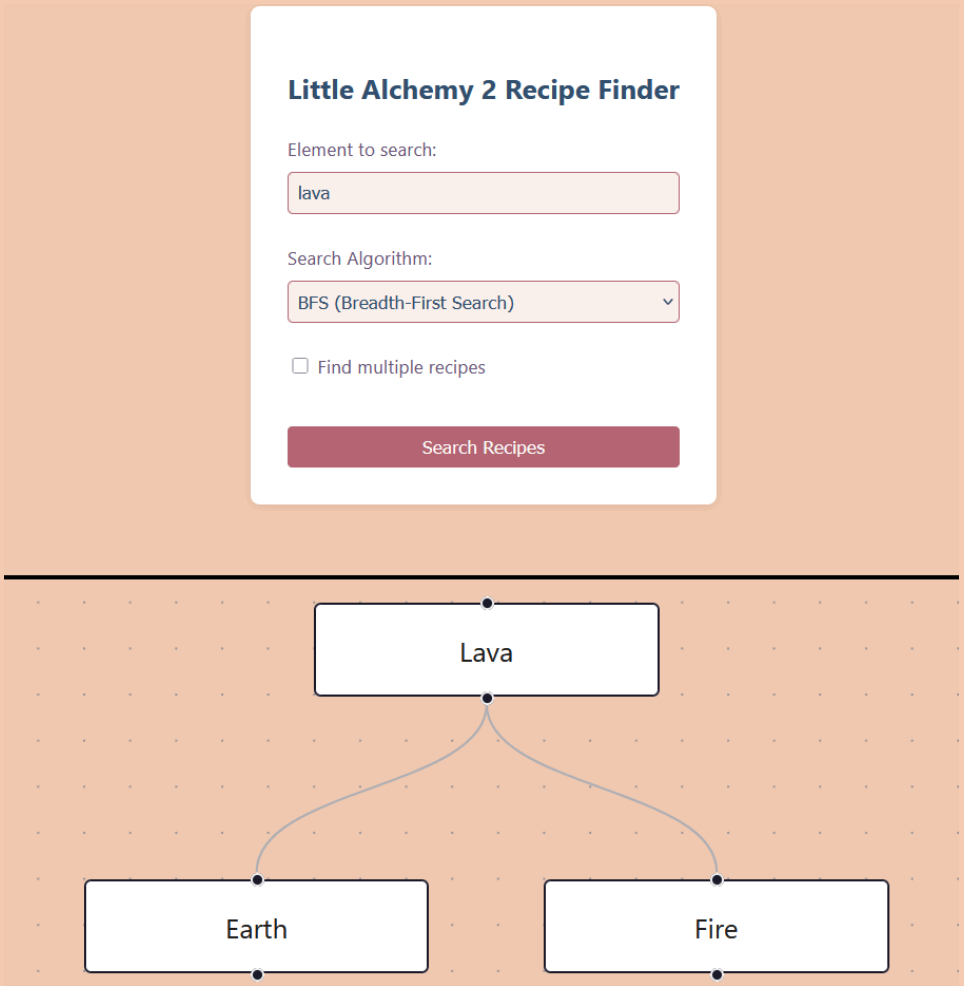
4.2 Tata Cara Penggunaan Program

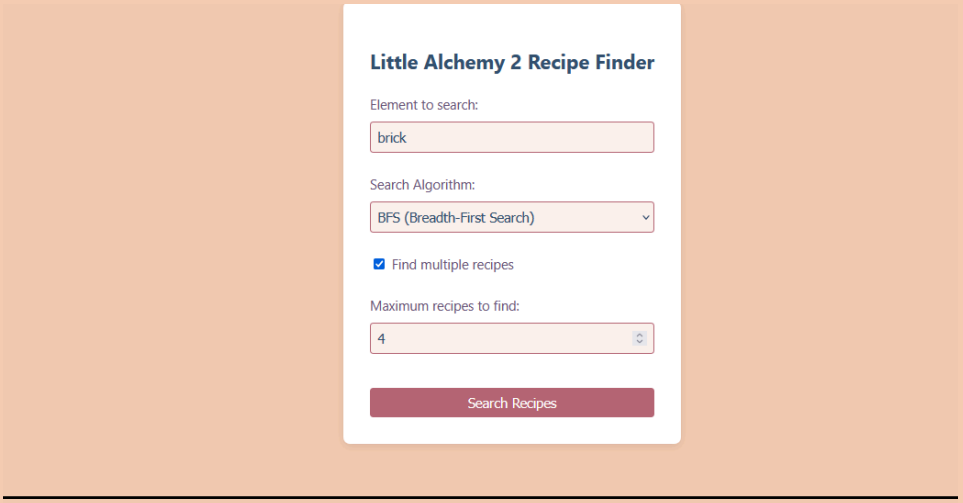
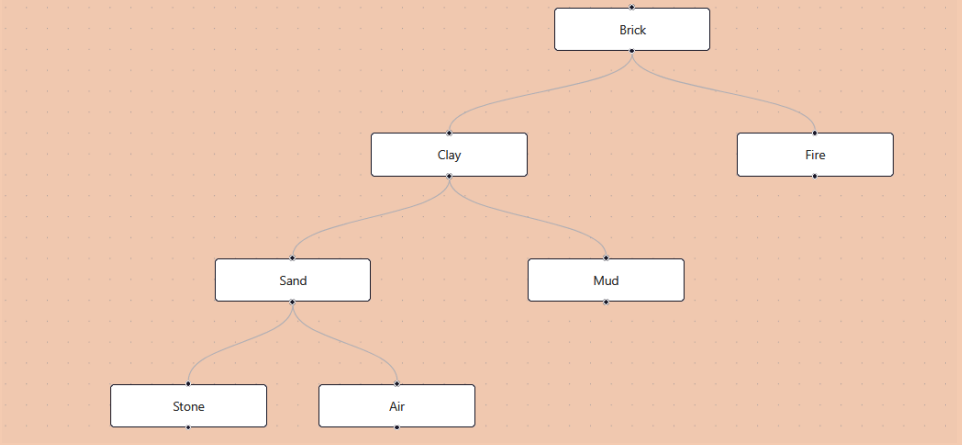
Untuk menjalankan program yang telah dibuat, pertama-tama pengguna perlu menjalankan backend dan frontend secara terpisah. Untuk menjalankan backend, pengguna harus masuk ke direktori backend dengan perintah **cd backend** melalui terminal, kemudian menjalankan perintah **go run main.go** untuk memulai backend server.

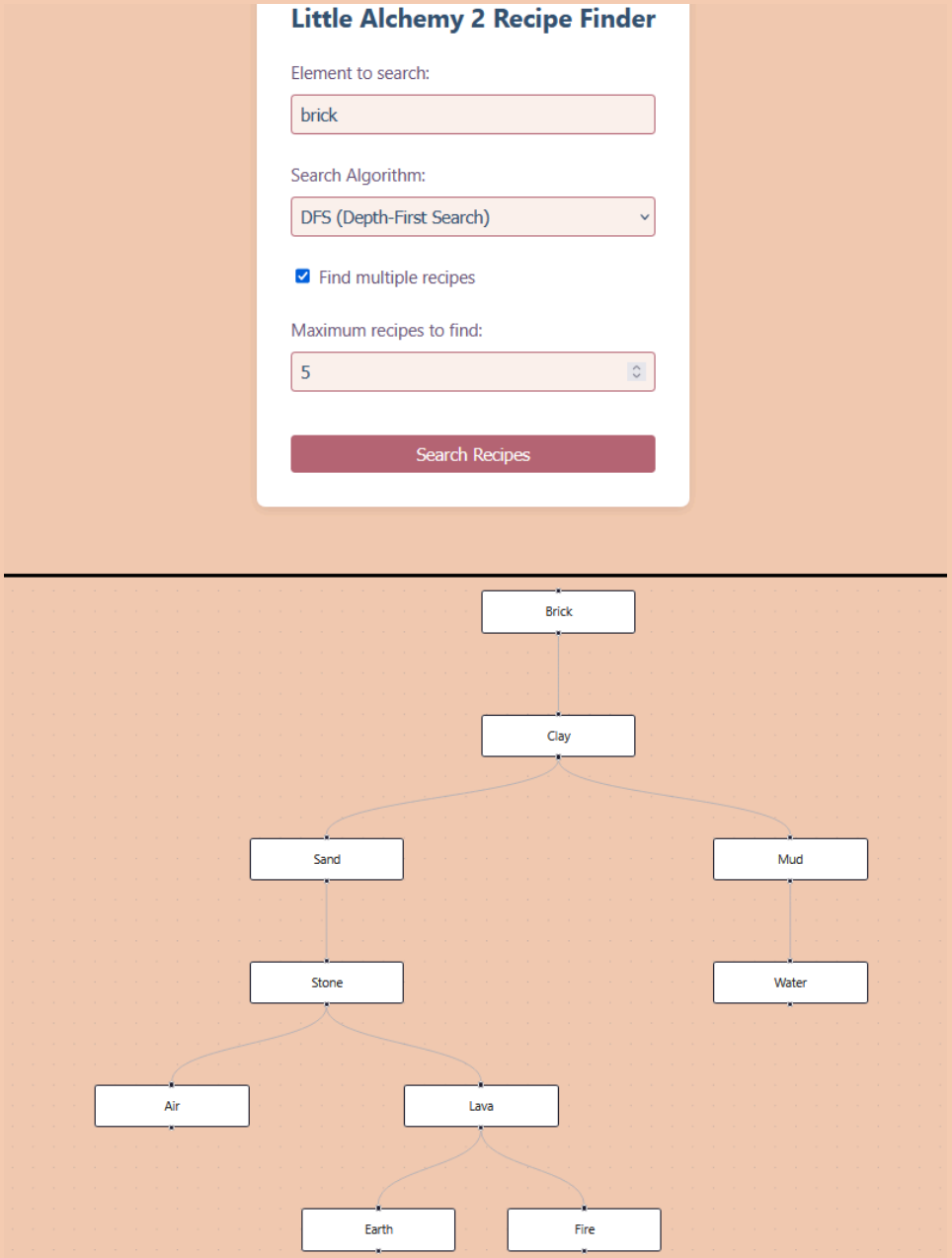
Setelah itu, untuk menjalankan *frontend*, pengguna harus masuk ke direktori *frontend* dengan perintah **cd frontend**, lalu menjalankan perintah **npm run build** untuk membangun aplikasi *frontend*. Setelah proses build selesai, jalankan perintah **npm run build** untuk memulai aplikasi *frontend* yang telah dibangun dan siap digunakan.

4.3 Hasil Pengujian

Pengujian ke-	Hasil Percobaan

1	 Algorithm = BFS, Nodes = 4855, Duration = 36855ms

2	  Algorithm = BFS, Nodes = 2482, Duration = 17782ms

3	 Algorithm = DFS, Nodes = 61, Duration = 6613ms

4.4 Analisis Hasil Pengujian

Hasil dari pengujian menunjukkan bahwa algoritma *Breadth-First Search* (BFS) secara umum mampu membangun pohon resep dengan struktur yang lebih baik dibandingkan algoritma lainnya, di mana sebagian besar node menampilkan kombinasi resep yang logis dan sesuai dengan data hasil scraping. BFS bekerja dengan menelusuri setiap level pohon secara sistematis, sehingga menghasilkan urutan pembangunan elemen yang lebih terstruktur. Namun demikian, hasilnya masih belum sepenuhnya sempurna. Terdapat beberapa kasus,

seperti elemen *Mud* dan *Stone*, yang seharusnya memiliki resep pembentuk namun tidak muncul dalam pohon. Hal ini menunjukkan bahwa meskipun BFS relatif lebih akurat, masih terdapat kekurangan dalam integrasi data dari hasil scraping atau dalam proses rekursi pembentukan pohon.

Pada *test case* pertama dan kedua, dapat dilihat bahwa lava terbuat dari *fire* (api) dan *earth* (bumi). *Brick* (batu bata) terbuat dari *clay* (tanah liat) dan *fire*, Clay terbuat dari *sand* (pasir) dan *mud* (lumpur), *sand* (pasir) terbuat dari *stone* (batu) dan *air* (udara).

Sebaliknya, saat menggunakan algoritma *Depth-First Search* (DFS), ditemukan kejanggalan pada hasil pohon yang dibentuk. Beberapa node dalam pohon hasil DFS hanya menampilkan satu child recipe atau bahkan struktur yang tidak lengkap, padahal seharusnya setiap node (yang bukan elemen dasar) memiliki kombinasi dua elemen pembentuk. Hal ini kemungkinan disebabkan oleh cara DFS memprioritaskan pencarian ke jalur terdalam terlebih dahulu, sehingga menyebabkan *state visited* berubah lebih awal dan mencegah eksplorasi kombinasi lainnya, yang pada akhirnya menghasilkan pohon yang tidak sepenuhnya akurat.

Pada *test case* ketiga, dapat dilihat bahwa *brick* hanya terbuat dari *clay*, padahal seharusnya *clay* dengan *fire*, *clay* dengan *sun*, *clay* dengan *stone*, *mud* dengan *fire*, dan *mud* dengan *sun*. Begitu juga dengan *element sand* dan *mud* yang hanya memiliki satu *node*.

Bab 5
Kesimpulan dan Saran

a. Kesimpulan

Program yang dibuat telah berhasil menjalankan fungsinya dengan baik dalam melakukan scraping, membangun *tree*, dan menampilkannya di *frontend*. Data yang diambil melalui proses *scraping* dapat diproses dengan benar untuk membentuk struktur pohon yang ditampilkan pada antarmuka pengguna. Namun, terdapat kesalahan pada backend yang perlu diperbaiki, dimana dalam beberapa kasus sebuah *node* hanya memiliki satu buah *recipe*, padahal seharusnya setiap node memiliki nol atau dua *recipe*. Hal ini menunjukkan bahwa meskipun secara umum program berjalan dengan baik, terdapat celah dalam logika pemrosesan data yang harus diperbaiki untuk memastikan bahwa struktur pohon terbentuk dengan benar di semua kasus.

Selama proses scraping dan pembangunan *tree*, terdapat beberapa kendala teknis yang cukup signifikan. Salah satunya adalah proses scraping yang dilakukan berulang-ulang pada setiap iterasi dalam membangun tree, yang menyebabkan proses berjalan lebih lama dari yang diharapkan. Hal ini dikarenakan setiap kali membangun sebuah *node*, program harus melakukan *scraping* kembali untuk mendapatkan data yang diperlukan, yang berakibat pada waktu eksekusi yang lebih lama. Optimasi dalam proses *scraping* dan pembentukan *tree* akan sangat diperlukan untuk mempercepat waktu eksekusi dan meningkatkan efisiensi program.

b. Saran

Untuk mengatasi permasalahan yang ditemukan dalam program, beberapa perbaikan perlu dilakukan. Pertama, terkait dengan kesalahan pada backend di mana sebuah node hanya memiliki satu recipe padahal seharusnya bisa memiliki nol atau dua, diperlukan pengecekan lebih teliti dalam logika pemrosesan data pada backend. Sebaiknya, dilakukan pemeriksaan lebih mendalam untuk memastikan bahwa setiap node memiliki jumlah recipe yang sesuai dengan kondisi yang diinginkan. Hal ini dapat dilakukan dengan memvalidasi data sebelum membentuk tree atau menambahkan mekanisme *fallback* jika terjadi kasus yang tidak sesuai dengan harapan.

Selain itu, untuk mempercepat proses *scraping* dan pembangunan *tree*, sebaiknya proses *scraping* tidak dilakukan berulang-ulang pada setiap iterasi. Salah satu solusi yang dapat diterapkan adalah dengan menyimpan hasil *scraping* dalam *cache* atau *database* sementara, sehingga data yang telah diambil tidak perlu diproses ulang di setiap iterasi. Dengan cara ini, waktu eksekusi dapat dipercepat dan program menjadi lebih efisien. Optimasi ini juga dapat mencakup pemrosesan data secara paralel atau asinkron, jika memungkinkan, untuk meminimalisir waktu tunggu selama proses pembangunan tree.

LAMPIRAN

Tautan *repository* GitHub : Tubes2_FullmetalJavascript

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, R. (2025). *Breadth/Depth First Search (BFS/DFS)*. Homepage Rinaldi Munir. Retrieved May 6, 2025, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf