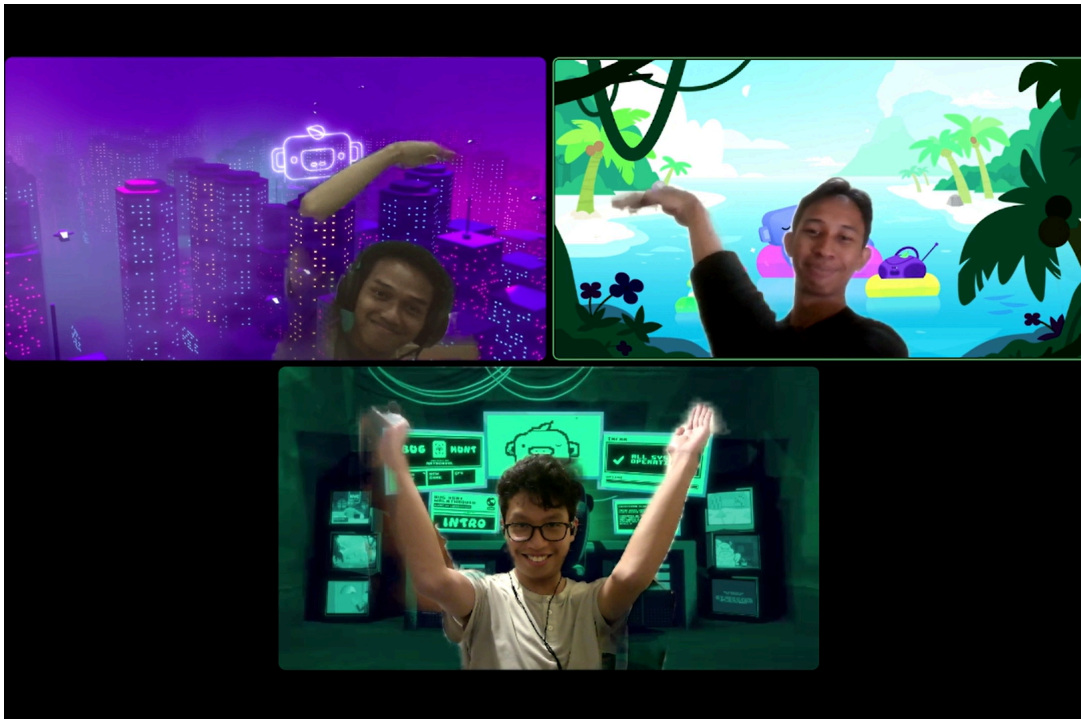


Tugas Besar 2

IF2211 Strategi Algoritma

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



Disusun oleh:

Timothy Niels Ruslim (10123053)
Rafa Abdussalam Danadyaksa (13523133)
Muhammad Nazih Najumudin (13523144)

Program Studi Teknik Informatika
Sekolah Teknik Elektro Dan Informatika
Institut Teknologi Bandung
Jl. Ganesa 10, Bandung 40132
2025

DAFTAR ISI

DAFTAR ISI	2
BAB I	4
1.1. Spesifikasi Wajib	5
1.2. Isi laporan	8
1.3. Penilaian	8
BAB II	10
2.1. Penjelajahan Graf	10
2.1.1. Algoritma Breadth First Search (BFS)	10
2.1.2. Algoritma Depth First Search (DFS)	10
2.2. Pohon Dinamis	11
2.3. Teori Aplikasi Web	11
BAB III	12
3.1. Langkah-langkah Pemecahan Masalah	12
3.2. Pemetaan Masalah	12
3.2.1. Pohon Ruang Status	12
3.2.1. BFS	14
3.2.2. DFS	14
3.3. Aplikasi Web	14
3.3.1. Fitur Fungsional	15
3.3.2. Arsitektur	15
BAB IV	16
4.1. Implementasi Scraper	16
4.2.1. Parser.go	16
4.2.1. Store.go	17
4.2.1. Scraper.go	17
4.2. Implementasi Backend	18
4.2.1. Repository Layer	19
4.2.2. HTTP Handler	19
4.2.3. Struktur Pohon Dinamis	21
4.2.4. Builder	26
4.2.5. BFS	27
4.2.6. DFS	30

4.3. Implementasi Frontend	32
4.3.1. Components	33
4.3.2. HomePage	37
4.3.3. Utils	39
4.3. Pengujian	41
BAB V	49
5.1. Kesimpulan	49
5.2. Saran	49
5.3. Refleksi	49
LAMPIRAN	50
Link Penting	50
Tabel Checkpoint	50
DAFTAR PUSTAKA	51

BAB I

DESKRIPSI TUGAS

Little Alchemy 2 merupakan permainan berbasis web atau aplikasi yang dikembangkan oleh Reclock 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.



Gambar 1.1 Little Alchemy 2
(sumber: www.thegamer.com)

Pada Tugas Besar 2 Strategi Algoritma ini, kami akan menyelesaikan permainan Little Alchemy 2 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 1.2 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*

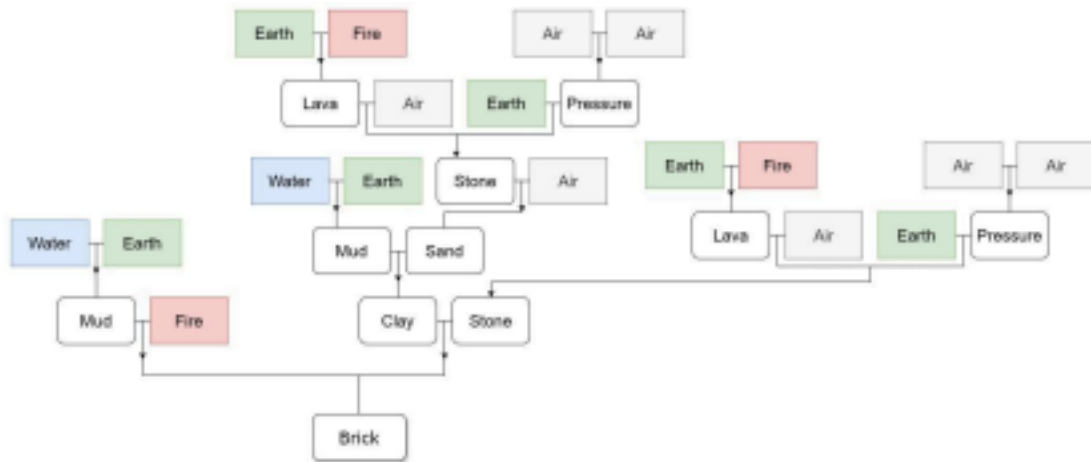
Untuk mendapatkan elemen turunan pemain dapat melakukan *combine* antara 2 elemen untuk menghasilkan elemen baru. Elemen turunan yang telah didapatkan dapat digunakan kembali oleh pemain untuk membentuk elemen lainnya.

1.1. Spesifikasi Wajib

- Buatlah aplikasi pencarian *recipe* elemen dalam permainan Little Alchemy 2 dengan menggunakan strategi **BFS dan DFS**.
- Tugas dikerjakan berkelompok dengan anggota **minimal 2 orang** dan **maksimal 3 orang**, boleh lintas kelas dan lintas kampus.
- Aplikasi berbasis **web**, untuk *frontend* dibangun menggunakan bahasa **Javascript** dengan *framework* **Next.js atau React.js**, dan untuk *backend* menggunakan bahasa **Golang**.
- Untuk *repository frontend* dan *backend* diperbolehkan **digabung** maupun **dipisah**.
- Untuk data elemen beserta resep dapat diperoleh dari *scraping* [website Fandom Little Alchemy 2](https://www.fandom.com/wiki/Little_Alchemy_2).
- Terdapat opsi pada *aplikasi* untuk **memilih algoritma** BFS atau DFS (juga *bidirectional* jika membuat bonus)
- Terdapat *toggle button* untuk memilih untuk menemukan **sebuah *recipe* terpendek** (*output* dengan rute terpendek) atau **mencari banyak *recipe* (*multiple recipe*)** menuju suatu elemen tertentu. Apabila pengguna ingin mencari banyak *recipe* maka terdapat cara bagi pengguna untuk **memasukkan parameter banyak *recipe*** maksimal yang ingin dicari. Aplikasi boleh mengeluarkan *recipe* apapun asalkan berbeda dan memenuhi

banyak yang diinginkan pengguna (apabila mungkin).

- Mode pencarian *multiple recipe* wajib dioptimasi menggunakan **multithreading**. • Aplikasi akan **memvisualisasikan recipe** yang ditemukan sebagai sebuah *tree* yang menunjukkan kombinasi elemen yang diperlukan dari elemen dasar. Agar lebih jelas perhatikan contoh berikut



Gambar 1.3 Contoh visualisasi *recipe* elemen

Gambar diatas menunjukkan contoh visualisasi *recipe* dari elemen *Brick*. Setiap elemen bersebelahan menunjukkan elemen yang perlu dikombinasikan. Amati bahwa *leaf* dari *tree* selalu berupa elemen dasar. Apabila dihitung, gambar diatas menunjukkan 5 buah *recipe* untuk *Brick* (karena *Brick* dapat dibentuk dengan kombinasi *Mud+Fire* atau *Clay+Stone*, begitu pula *Stone* yang dapat dibentuk oleh kombinasi *Lava+Air* atau *Earth+Pressure*). Visualisasi pada aplikasi tidak perlu persis seperti contoh diatas, tetapi pastikan bahwa *recipe* ditampilkan dengan jelas.

- Aplikasi juga menampilkan **waktu pencarian** serta **banyak node** yang dikunjungi.

Spesifikasi Bonus

- **(maks 5)** Membuat video tentang aplikasi BFS dan DFS pada permainan Little Alchemy 2 di Youtube. Video dibuat harus memiliki audio dan menampilkan wajah dari setiap anggota kelompok. Untuk contoh video tubes stima tahun-tahun sebelumnya dapat dilihat di Youtube dengan kata kunci “Tubes Stima”, “strategi algoritma”, “Tugas besar stima”, dll. **Semakin menarik video, maka semakin banyak poin yang diberikan.**
- **(maks 3)** Aplikasi dijalankan menggunakan **Docker** baik untuk *frontend* maupun *backend*.
- **(maks 3)** Aplikasi **di-deploy** ke aplikasi *deployment* (aplikasi *deployment* bebas) agar bisa diakses secara daring
- **(maks 3)** Menambahkan algoritma bukan hanya DFS dan BFS, tetapi juga strategi **bidirectional**

- **(maks 6)** Aplikasi memiliki fitur *Live Update* visualisasi *recipe* selama proses pencarian. *Tree* visualisasi akan dibangun bertahap secara *real time* sesuai dengan progress pencarian. Tambahkan *delay* pada Live Update karena pencarian dapat berjalan dengan sangat cepat.
- Jika terdapat kesulitan selama mengerjakan tugas besar sehingga memerlukan bimbingan, maka dapat melakukan asistensi tugas besar kepada asisten (opsional). Dengan catatan asistensi hanya bersifat membimbing, bukan memberikan “jawaban”.
- Terdapat demo dengan asisten untuk mendemonstrasikan aplikasi yang telah dibuat. Pengumuman mengenai demo akan diberitahukan lebih lanjut oleh asisten. • Setiap kelompok harap mengisi nama kelompok dan anggotanya pada link berikut, paling lambat **Kamis, 24 April pukul 22.11 WIB**.
- Diwajibkan untuk memilih asisten meskipun tidak melakukan asistensi, karena asisten yang dipilih akan menjadi asisten saat asistensi (opsional) dan demo tugas besar. Pemilihan asisten dapat dilakukan pada link berikut, paling lambat **Kamis, 24 April 2025 pukul 22.11 WIB**.
- Program disimpan dalam repository yang bernama **Tubes2_NamaKelompok** (bila digabung) dan **Tubes2_FE/BE_NamaKelompok** (bila dipisah) dengan nama kelompok sesuai dengan yang di *sheets* diatas. Berikut merupakan struktur dari isi repository tersebut:
 - a. Folder src berisi **program yang dapat dijalankan**
 - b. Folder doc berisi laporan tugas besar dengan format **NamaKelompok.pdf** c. README untuk tata cara penggunaan yang minimal berisi:
 - i. Penjelasan singkat algoritma DFS dan BFS yang diimplementasikan
 - ii. Requirement program dan instalasi tertentu bila ada
 - iii. Command atau langkah-langkah dalam meng-compile atau build program
 - iv. Author (identitas pembuat)
- Sangat disarankan untuk menggunakan [*semantic commit*](#). Buatlah release dengan format **v1.x** dengan x adalah nomor revisi dimulai dari revisi 0. Contoh v1.0 untuk release pertama, v1.1 untuk revisi selanjutnya.
- Pastikan untuk membuat repository bersifat **Public** paling lambat **H+1 deadline (1 hari setelah deadline)**. Sebelum deadline repository harus bersifat **Private**.
- Laporan dikumpulkan hari **Senin, 12 Mei 2025** pada alamat Google Form berikut paling lambat **pukul 22.11 WIB**:
<https://bit.ly/tubes2stima25>
- **PERINGATAN: Keterlambatan akan mengurangi nilai sebanyak 1 poin untuk setiap menit keterlambatan.**
- Adapun pertanyaan terkait tugas besar ini bisa disampaikan melalui QnA berikut:
<https://bit.ly/QnA-Stima-25>.

1.2. Isi laporan

- **Cover:** Cover laporan ada foto anggota kelompok (foto bertiga). Foto ini menggantikan logo “gajah” ganesha.
- **Bab 1:** Deskripsi tugas (dapat menyalin spesifikasi tugas ini).
- **Bab 2:** Landasan Teori.
 - Dasar teori (Penjelajahan Graf serta algoritma *Breadth First Search* dan *Depth First Search* secara umum serta bonus).
 - Penjelasan singkat mengenai aplikasi web yang dibangun.
- **Bab 3:** Analisis Pemecahan Masalah.
 - Langkah-langkah pemecahan masalah.
 - Proses pemetaan masalah menjadi elemen-elemen algoritma DFS dan BFS.
 - Fitur fungsional dan arsitektur aplikasi web yang dibangun.
 - Contoh ilustrasi kasus.
- **Bab 4:** Implementasi dan pengujian.
 - Spesifikasi teknis program (struktur data, fungsi, dan prosedur yang dibangun).
 - Penjelasan tata cara penggunaan program (*interface* program, fitur-fitur yang disediakan program, dan sebagainya).
 - Hasil pengujian **minimal 3** buah elemen dalam bentuk *screenshot* antarmuka. Variasikan setiap kasus dengan berbagai fitur yang diimplementasikan (Algoritma serta banyak *recipe* yang dicari).
 - Analisis hasil pengujian.
- **Bab 5:** Kesimpulan, Saran, dan Refleksi tentang Tugas Besar 2.
- **Lampiran:** Tautan *repository* GitHub (dan video jika membuat)
- **Daftar Pustaka**

Keterangan laporan:

1. Laporan ditulis dalam bahasa Indonesia yang baik dan benar.
2. Laporan mengikuti format pada *section* “Isi laporan” dengan baik dan benar.
3. Identitas per halaman harus jelas (misalnya : halaman, kode kuliah).

1.3. Penilaian

1. Bagian 1: Laporan (40%)

- a. Langkah-langkah pemecahan masalah, proses pemetaan masalah, fitur fungsional dan arsitektur aplikasi web yang dibangun, dan contoh ilustrasi kasus. (20 %)
- b. Spesifikasi teknis program (struktur data, fungsi, dan prosedur yang dibangun), penjelasan tata cara penggunaan program (*interface* program, fitur-fitur yang disediakan program, dan sebagainya), hasil pengujian (*screenshot* antarmuka dan variasi pengujian kasus berdasarkan fitur aplikasi), dan analisis hasil pengujian. (15 %)

- c. Kelengkapan komponen-komponen pada laporan dan README. (5 %)
- 2. Bagian 2: Implementasi Program dan Demo (60%)**
 - a. Kecocokan output dari test case yang dimiliki Asisten. (25 %)
 - b. Kebenaran algoritma *Depth First Search* dan *Breadth First Search*, sesuai dengan apa yang telah diajarkan di kelas. (15%)
 - c. Pemahaman tugas besar dan algoritma yang telah dibuat oleh masing-masing anggota. (10%)
 - d. Keberhasilan input dan output, sesuai dengan komponen-komponen yang ada pada spesifikasi. (5%)
 - e. Modularitas/keterbacaan penulisan program. (5%)
- 3. Bagian 3: Komponen Bonus (Maksimal 20 Poin)**
 - a. Program dapat menampilkan *Live Update* visualisasi secara *real time*. (**bonus** maksimal 6 poin)
 - b. Aplikasi dapat melakukan pencarian dengan strategi Bidirectional. (**bonus** maksimal 3 poin)
 - c. Aplikasi di-deploy dan dapat diakses melalui internet. (**bonus** maksimal 3 poin)
 - d. Program dijalankan menggunakan Docker baik untuk *frontend* maupun *backend*. (**bonus** maksimal 3 poin)
 - e. Bonus dalam membuat video kelompok. (bonus maksimal 5 poin)

BAB II

TEORI SINGKAT

2.1. Penjelajahan Graf

Graf adalah struktur yang berisi sekumpulan objek yang direpresentasikan sebagai simpul dan hubungan antara objek-objek tersebut yang direpresentasikan sebagai sisi. Struktur ini sangat berguna untuk merepresentasikan permasalahan dengan cara yang lebih matematis, sehingga dapat dianalisis dengan lebih mudah. Salah satu hal yang sering ingin dilakukan adalah penjelajahan graf, yaitu proses mengunjungi simpul-simpul pada graf berdasarkan sisi yang ada dan aturan yang diinginkan. Sebagai contoh, jika graf merepresentasikan jaringan jalan, penjelajahan graf dapat melibatkan pencarian rute dari suatu lokasi ke lokasi lain, sehingga dapat digunakan untuk memodelkan sistem navigasi. Umumnya, ada dua cara paling terkenal untuk menjelajahi suatu graf: *Breadth First Search* (BFS) dan *Depth First Search* (DFS).

2.1.1. Algoritma *Breadth First Search* (BFS)

Algoritma *Breadth First Search* (BFS) merupakan algoritma traversal graf yang mengunjungi simpul-simpul secara melebar, dimulai dari simpul awal, lalu menjelajahi semua simpul tetangganya sebelum melanjutkan ke level yang lebih dalam. BFS menggunakan *queue* untuk menyimpan simpul-simpul yang akan dikunjungi untuk memastikan bahwa setiap simpul dikunjungi berdasarkan urutan kedalaman. BFS merupakan algoritma yang lengkap, artinya BFS akan selalu menemukan solusi jika solusi itu ada, serta diberikan waktu dan *resource* yang cukup. Kelemahan algoritma BFS adalah kompleksitas ruangnya yang tinggi, yaitu $O(b^d)$, dengan b adalah *branching factor*, dan d adalah kedalaman solusi. Tingginya kompleksitas ruang BFS disebabkan algoritma ini harus menyimpan setiap simpul yang hendak dikunjungi dalam *queue* (yang disimpan di memori) sebelum melanjutkan ke level berikutnya.

2.1.2. Algoritma *Depth First Search* (DFS)

Algoritma *Depth First Search* (DFS) merupakan algoritma traversal graf yang mengunjungi simpul-simpul secara mendalam dengan mengeksplorasi satu cabang sejauh mungkin sebelum mencapai daun, lalu mundur (*backtrack*) dan mencoba cabang lainnya. Algoritma ini menggunakan *stack* atau rekursi untuk mengeksplorasi simpul-simpul. Hal inilah yang membuat DFS memiliki kompleksitas ruang yang lebih rendah dibanding BFS, yaitu $O(bm)$, dengan b adalah *branching factor*, dan m adalah kedalaman maksimum pohon. Meskipun memiliki kompleksitas ruang yang lebih baik dibanding BFS, algoritma DFS tidak selalu optimal karena penelusuran mungkin saja menemukan solusi yang lebih panjang sebelum menemukan solusi terpendek. Selain itu, algoritma DFS juga tidak lengkap, karena jika graf memiliki kedalaman tak terbatas DFS tidak dapat bekerja, kecuali dengan penanganan khusus. Keunggulan DFS yang mengeksplorasi pohon secara mendalam, cocok untuk digunakan dalam pencarian rute labirin

dan permainan seperti catur atau sudoku. Algoritma DFS juga menjadi dasar algoritma *backtracking*.

2.2. Pohon Dinamis

Berdasarkan penggunaan traversal graf, pohon dapat dibagi menjadi dua jenis, yaitu pohon statis dan pohon dinamis. Pohon statis merupakan struktur pohon yang sudah dibentuk sebelumnya sebelum proses traversal graf dilakukan. Pohon statis direpresentasikan sebagai graf yang sudah diketahui seluruh simpul dan sisinya yang bersifat tetap. Contoh dari pohon statis adalah peta rute tetap seperti kota-kota dalam peta. Sedangkan, pohon dinamis merupakan pohon yang dibangun secara bertahap selama proses traversal graf berlangsung. Pohon dinamis tidak diketahui sebelumnya, dan dikembangkan berdasarkan aturan tertentu yang diterapkan pada simpul-simpul yang sedang dieksplorasi. Contoh dari pohon dinamis adalah pohon ruang status permainan, serta penelusuran web. Pohon dinamis berguna dalam pemecahan masalah yang memerlukan eksplorasi bertahap seperti labirin dan permainan. Pohon dinamis memiliki fleksibilitas tinggi untuk menyelesaikan masalah kompleks, tetapi juga memerlukan lebih banyak komputasi, karena pembentukan pohon dilakukan secara *real-time*.

2.3. Teori Aplikasi Web

Aplikasi web merupakan perangkat lunak yang dijalankan melalui web browser dan diakses menggunakan jaringan internet atau intranet tanpa bergantung pada sistem operasi tertentu. Hal ini juga memungkinkan aplikasi web digunakan tanpa perlu menginstall aplikasi secara lokal pada perangkat, serta pengguna dapat selalu menggunakan versi terbaru dari aplikasi secara *remote*. Aplikasi web sangat efektif untuk memenuhi kebutuhan akses yang membutuhkan fleksibilitas tinggi seperti dalam layanan pendidikan, bisnis, dan masih banyak lagi. Salah satu *framework* yang dapat digunakan untuk membangun aplikasi web adalah menggunakan React.

React merupakan *library* JavaScript yang dikembangkan oleh Facebook untuk membangun UI pada aplikasi web dan mobile. React memfokuskan pada komponen-komponen UI yang dapat digunakan ulang, sehingga memudahkan pengembangan aplikasi dengan tampilan yang kompleks dan performa tinggi. React menggunakan Virtual DOM dan JSX (JavaScript XML) sebagai sintaks yang memadukan JavaScript dan HTML. React banyak digunakan dalam pengembangan aplikasi modern seperti Instagram dan Netflix.

BAB III

ANALISIS PEMECAHAN MASALAH

3.1. Langkah-langkah Pemecahan Masalah

Kami diberikan permasalahan berikut: mencari recipe untuk suatu elemen di Little Alchemy 2. Dengan permasalahan ini, kami juga memiliki beberapa kendala.

1. Kendala Frontend
 - a. Menggunakan bahasa Javascript
 - b. Menggunakan *framework* React.js atau Next.js
 - c. Menerima masukan berupa jumlah recipe yang diinginkan
 - d. Dapat memvisualisasikan recipe sebagai suatu pohon
 - e. Menampilkan informasi seperti simpul yang dijelajahi dan waktu pencarian
2. Kendala Backend
 - a. Menggunakan bahasa Go
 - b. Melakukan *scraping* data recipe dari Wiki Little Alchemy 2
 - c. Membuat data struktur pohon untuk merepresentasikan recipe suatu elemen
 - d. Dapat mengimplementasikan BFS dan DFS

Berdasarkan masalah dan kendala tersebut, kami membagi tugas besar ini menjadi tiga tahap: pembuatan *scraper* data recipe, pemetaan masalah mencari recipe untuk elemen di Little Alchemy 2 menjadi suatu masalah pohon dinamis yang dapat diselesaikan dengan algoritma BFS dan DFS, dan perancangan desain aplikasi web yang dapat menjadi *frontend* baik bagi user yang ingin mencari recipe dari suatu elemen. Proses *scraping* sebenarnya jelas dan mudah, sehingga bisa dilihat langsung implementasinya di 4.1. Namun, proses memecahkan masalah ini melalui pemetaan komponen di *backend* maupun rekayasa desain aplikasi web di *frontend* memerlukan perawatan yang lebih.

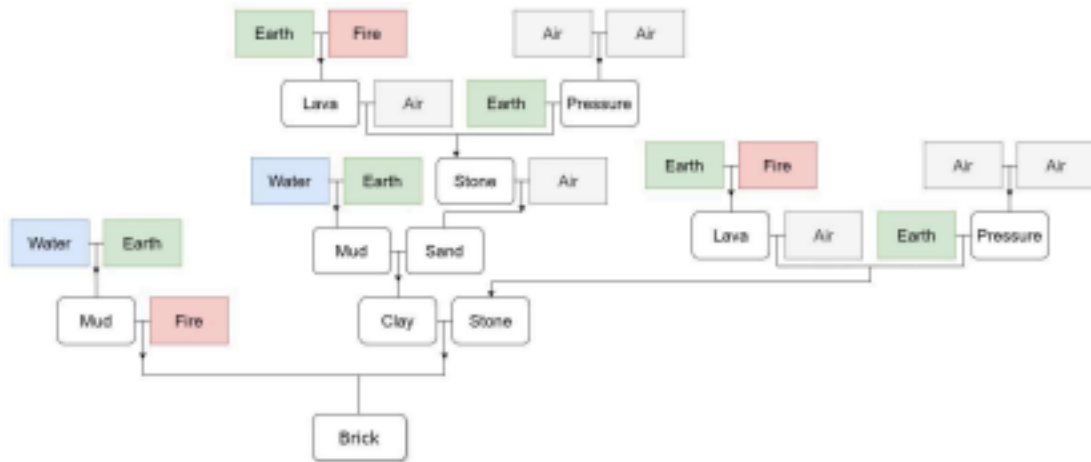
3.2. Pemetaan Masalah

Pertama, akan dibahas pemetaan masalah mencari recipe suatu elemen di Little Alchemy 2 menjadi elemen-elemen algoritmis. Kami akan menggunakan konsep pohon ruang status, seperti yang dijelaskan di landasan teori, yang dibentuk secara dinamis menggunakan algoritma BFS dan DFS.

3.2.1. Pohon Ruang Status

Di Little Alchemy 2, penemuan suatu elemen dapat dinyatakan sebagai suatu keadaan atau status. Lalu, penggabungan dua elemen dapat dianggap sebagai aksi transisi antar status. Dalam pohon ruang status, ini berarti kita bisa merepresentasikan suatu elemen sebagai simpul dan

suatu penggabungan sebagai sisi. Ini dapat menghasilkan struktur pohon seperti yang ada pada spesifikasi tugas besar ini.



Gambar 3.1 Pohon *Recipe* Elemen (dari spesifikasi)

Perhatikan bahwa simpul akar merepresentasikan elemen yang resepnya ingin dicari, sedangkan simpul daun adalah elemen primer. Walaupun pemetaan ini sudah cukup bagus dan terstruktur, kami juga memikirkan kepraktisannya. Implementasi sisi yang merepresentasikan penggabungan elemen dengan pendekatan pohon seperti ini akan lumayan memusingkan, karena terasa kurang modular. Misalnya, resep untuk “Stone” dapat datang dari “Lava + Air” dan “Earth + Pressure”. Kalau begitu, apakah anak simpul dari “Stone” menjadi “Lava”, “Air”, “Earth”, “Pressure”? Mungkin iya, tetapi bagaimana membedakannya untuk mengetahui penggabungan yang benar? Jika dikodifikasikan dalam larik bertingkat $[[\text{“Lava”, “Air”}], [\text{“Earth”, “Pressure”}]]$, kami menjadi bekerja dengan matriks padahal tujuannya untuk membuat pohon sederhana. Oleh karena itu, dalam implementasi kami, akan dibuat suatu simpul recipe yang menyatakan penggabungan secara khusus, yang terpisah dari simpul elemen. Untuk lebih jelasnya, dapat dibaca di implementasi pohon ini di 4.2.3.

Hal lain yang harus dibicarakan adalah makna *solusi* dari pohon ruang status. Pada banyak game, simpul daun adalah suatu solusi. Tetapi, dalam pohon recipe untuk Little Alchemy 2, suatu solusi adalah *subtree* yang merepresentasikan suatu resep sah dari elemen. Bagaimana cara menghitung jumlah recipe pada suatu pohon? Misalkan cara membuat suatu elemen adalah b , dan elemen tersebut memiliki n resep r_1, r_2, \dots, r_n , dengan resep ke- k yaitu r_k menyatakan penggabungan dua elemen unsur yang memiliki jumlah recipe $b_k^{(1)}$ dan $b_k^{(2)}$ berturut-turut. Maka, menggunakan aturan perkalian dan penjumlahan *counting* (dari Matematika Diskrit), cukup mudah untuk melihat bahwa

$$b = \sum_{k=1}^n b_k^{(1)} \cdot b_k^{(2)}.$$

Rumus dapat kita gunakan untuk melihat apakah suatu pohon sudah mengandung *subtree* sesuai yang menjadi solusi.

3.2.1. BFS

Breadth First Search (BFS) dalam pembentukan pohon ruang status secara dinamis bekerja dengan pertama mencari semua aksi transisi yang mungkin dari suatu level. Untuk pohon recipe yang dijelaskan tadi, ini berarti mencari dulu semua recipe yang mungkin untuk membuat semua elemen pada kedalaman tertentu. Sebagai contoh, perhatikan Gambar 3.1 di atas. Untuk membangun pohon dinamis yang berakar di “Brick”, BFS akan pertama mencari semua recipe untuk “Brick”, lalu menjelajahi semua elemen unsurnya. Jadi, akan dikunjungi simpul “Mud”, “Fire”, “Clay”, dan “Stone” terlebih dahulu di kedalaman 1. Berikutnya, BFS akan bekerja per level. Jadi, urutan kunjungan berikutnya adalah “Water”, “Earth”, “Mud”, “Sand”, “Lava”, “Air”, “Earth”, dan “Pressure” di kedalaman 2. Perhatikan bahwa BFS akan menghasilkan pohon yang sangat lebar, dan karena suatu solusi berupa pohon recipe yang mengandung jumlah recipe sesuai yang diinginkan, ini akan membutuhkan kedalaman minimal sama dengan tier dari elemen. Oleh karena itu, BFS dapat mengunjungi sangat banyak simpul sebelum mendapatkan solusi. Tetapi, misalkan tidak dibatasi bahwa tier dari elemen unsur harus lebih rendah, BFS setidaknya akan menghasilkan solusi yang terpendek.

3.2.2. DFS

Depth First Search (DFS) dalam pembentukan pohon ruang status secara dinamis bekerja dengan mendalami suatu *branch* pada pohon hingga mencapai simpul daun sebelum melakukan runut balik. Untuk pohon recipe yang dijelaskan tadi, ini berarti mendalami salah satu resep dari elemen hingga mencapai unsur primernya, sebelum kembali mencari unsur lainnya. Sebagai contoh, dengan Gambar 3.1 yang sama, misal ingin dibangun pohon untuk “Brick”. DFS akan melihat recipe pertama, yaitu “Mud + Fire”, dan menelusuri unsur pertamanya, yaitu “Mud”. Lalu, hal yang sama akan dilakukan untuk “Mud”. Jadi, urutan kunjungannya untuk *branch* kiri yang pertama adalah “Mud”, “Water”, “Earth”, dan “Fire”. Kemudian, algoritma akan runut balik ke recipe kedua untuk “Brick”, dan mengulangi proses ini pada *branch* tersebut. Maka, DFS akan melanjutkan dengan mengunjungi “Clay”, “Mud”, “Water”, “Earth”, “Sand”, “Stone”, “Lava”, “Earth”, dan seterusnya. Algoritma DFS umumnya akan mencapai solusi lebih cepat dalam konstruksi pohon resep untuk Little Alchemy 2 ini. Alasannya adalah bahwa DFS tidak perlu mengunjungi semua simpul pada suatu level (yang biasanya sangat besar), sebelum memperoleh solusi. Tetapi, jika tidak dibatasi bahwa tier dari elemen unsur harus lebih rendah, ini berarti DFS dapat terjebak dalam siklus resep, karena pohon menjadi tidak terbatas.

3.3. Aplikasi Web

Aplikasi web dibangun menggunakan bahasa pemrograman JavaScript dengan menggunakan *framework* React. Penggunaan JavaScript dan React dipilih karena dapat memberikan tampilan

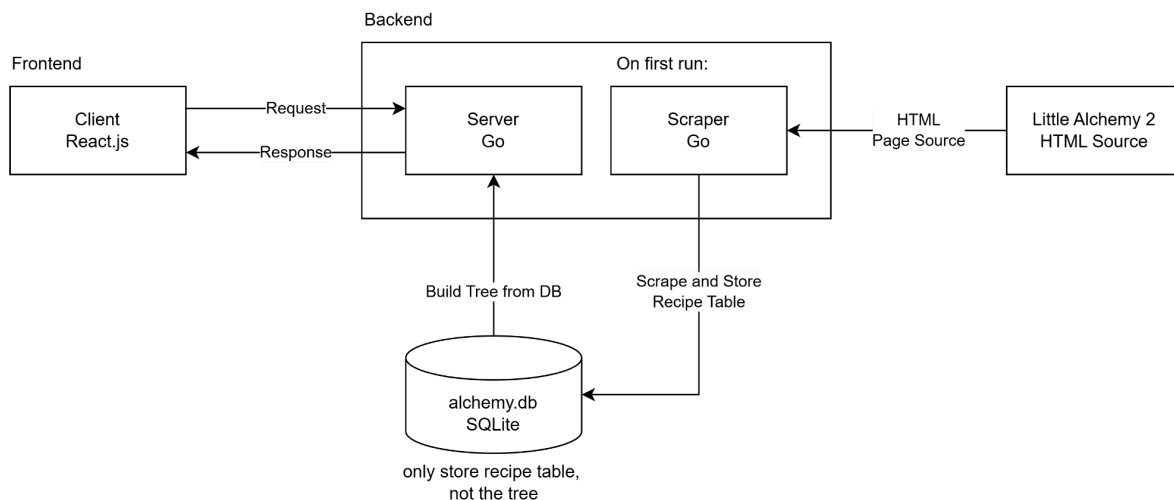
UI dan performa *frontend* yang baik, dengan kemudahan dalam pengembangan dan pemahaman penggunaannya.

3.3.1. Fitur Fungsional

Aplikasi memiliki beberapa fitur fungsional untuk menyelesaikan permasalahan. Pertama, pengguna dapat memasukkan input berupa nama elemen yang hendak dicari penyelesaiannya. Pengguna juga dapat memilih algoritma yang hendak digunakan (BFS atau DFS), mode pencarian (*singlepath recipe* atau *multipath recipe*) serta memasukkan banyak resep maksimal apabila menggunakan mode *multipath recipe*. Setelah menekan tombol search, pengguna akan mendapatkan hasil penyelesaian dalam bentuk rangkuman statistik pencarian, serta hasil pohon yang dibentuk. Pengguna dapat melakukan reset untuk memasukkan input baru.

3.3.2. Arsitektur

Arsitektur yang digunakan adalah *Client-Server Architecture*, yaitu arsitektur dengan membagi dua pekerjaan *frontend* (*client*) dan *backend* (*server*) yang saling berkomunikasi dengan API (*Application Programming Interface*). Pada tugas ini, *client* diimplementasi dengan React, dan *server* diimplementasi dengan Go. *Client* React berjalan pada port 3000, dan seluruh pemrosesan *request* pencarian solusi dilakukan pada *server* Go pada port 8080, serta menggunakan SQLite alchemy.db sebagai databasenya. Adapun penggunaan SQLite dipilih karena web tidak membutuhkan kompleksitas basis data yang tinggi.



Sumber: Dokumen Penulis

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1. Implementasi Scraper

Data *recipe* elemen-elemen diperoleh dengan men-*scrape* website wiki Little Alchemy 2 menggunakan bahasa Go. Dibuat data struktur Recipe untuk menyimpan data *recipe*.

Atribut	Jenis	Deskripsi
Element	string	Elemen yang <i>recipe</i> nyatakan.
Item1	string	Elemen pertama pada <i>recipe</i> .
Item2	string	Elemen kedua pada <i>recipe</i> .
IsPrimary	bool	Apakah elemen primer.

Struktur ini dibantu metode `isPrimary` yang mengecek apakah elemen primer.

fungsi <code>isPrimary</code>
Input: <code>element</code> Output: boolean <code>isPrimary</code>
<pre>function isPrimary: if element is in [Earth, Air, Fire, Water, Time]: return true else: return false</pre>

Ada juga data struktur hashmap `ElementTierMap` untuk menyimpan tier dari suatu elemen.

4.2.1. Parser.go

Ini mengandung fungsi utama yang melakukan *scraping*-nya yaitu `ParseHTML`.

fungsi <code>ParseHTML</code>
Input: <code>document</code> Output: list of <code>Recipe</code> , <code>ElementTierMap</code>
<pre>function ParseHTML: initialize empty list of recipes initialize empty map of element tiers set current tier to -1 for each section header in the document: get the section ID from the header</pre>

fungsi ParseHTML

```
if the section is a starting or special element:
    set current tier to 0
else if the section ID starts with "Tier_":
    extract the tier number from the ID
    set current tier accordingly
else:
    skip this section

find the first table after this header
if no table found:
    skip to next section

for each row in the table (skip the first row):
    get the element name from the first column
    if current tier is valid:
        record the tier of this element

    look for recipes in the second column:
        for each item listed:
            if the item contains '+':
                split it into two parts
                create a recipe with those two parts
                add the recipe to the list

    if no recipe was found or if the element is primary:
        add a recipe with no ingredients (special element)

return the list of recipes and the tier map
```

4.2.1. Store.go

Ini adalah struktur DataStore untuk melakukan penyimpanan *recipe* yang diperoleh dari parser. Berikut metode-metode (prosedur) yang dimilikinya.

Metode	Parameter	Deskripsi
SaveToCSV	<code>recipes []Recipe</code> <code>csvPath string</code>	Simpan <i>recipes</i> ke suatu file csv.
SaveToDB	<code>recipes []Recipe</code> <code>dbPath string</code>	Simpan <i>recipes</i> ke suatu SQLite database.
SaveMap	<code>tiers ElementTierMap</code> <code>jsonPath string</code>	Simpan tier dari elemen pada file json.

4.2.1. Scraper.go

Ini adalah struktur Scraper untuk menyatukan proses parse dan store secara modular menggunakan metode scrape-nya.

prosedur Scrape

Input: htmlPath, csvPath, dbPath, jsonPath **string**

Output: -

procedure Scrape:

open the **HTML file** at htmlPath
create a **document** from the **HTML file**
call **ParseHTML** to get recipes and element tiers

call **SaveToCSV** with recipes and csvPath
call **SaveToDB** with recipes and dbPath
call **SaveMap** with element tiers and jsonPath

procedure complete

4.2. Implementasi Backend

Backend dari aplikasi web menggunakan bahasa Go sepenuhnya juga. Dibuat struktur pohon serta simpulnya dan algoritma untuk mengkonstruksikan berdasarkan BFS atau DFS. Lalu, ada juga lapisan repository untuk mengambil *recipe* atau tier elemen dari file csv, db, atau json sebelumnya. Terakhir, ada HTTP handler untuk membuat endpoint API. Berikut struktur direktori dari *backend*.

```
backend/
├── cmd/
│   ├── api/
│   │   └── main.go
│   └── scraper/
│       └── main.go
├── data/
│   ├── alchemy.csv
│   ├── alchemy.db
│   ├── Elements (Little Alchemy 2) _ Little Alchemy Wiki _ Fandom.html
│   └── tiers.json
├── internal/
│   ├── handler/
│   │   └── http.go
│   ├── model/
│   │   ├── data-structures.go
│   │   ├── recipe.go
│   │   ├── tree-node.go
│   │   └── tree.go
│   ├── repo/
│   │   └── repo.go
│   ├── scraper/
│   │   ├── parser.go
│   │   ├── scrape.go
│   │   └── store.go
│   └── tree/
│       ├── bfs-builder.go
│       ├── builder.go
│       └── dfs-builder.go
├── Dockerfile
├── go.mod
├── go.sum
├── run-api.cmd
└── run-scraper.cmd
```

4.2.1. Repository Layer

Dibuat data struktur `RecipeRepository` untuk mengurus pengambilan data recipe dan tier dari elemen.

Atribut	Jenis	Deskripsi
db	sql.DB	File database yang digunakan (jika ada).
csvPath	string	File csv yang digunakan (jika ada).
tiers	ElementTierMap	Hashmap yang menyimpan tier elemen.
mode	string	Apakah menggunakan database atau csv.

Berikut juga metode-metode pada repository. Perlu dicatat bahwa recipe yang diperoleh menggunakan repository ini hanya mengandung recipe dengan unsur elemen dengan **tier lebih rendah**, sesuai interpretasi jawaban asisten pada QnA tugas ini.

Metode	Input	Output	Deskripsi
GetRecipesFor	element string	[]Recipe	Mengambil recipe untuk elemen.
getFromDB	element string	[]Recipe	Cari recipe dari database.
getFromCSV	element string	[]Recipe	Cari recipe dari csv.
getTierMap	jsonPath string	ElementTierMap	Ambil tier elemen dari json.

4.2.2. HTTP Handler

API dibuat menggunakan package `gin`, dengan membuat suatu `gin.Engine` yaitu router yang mengurus request HTTP. API memiliki *query parameter* sebagai berikut.

1. element: elemen yang recipe-nya ingin dicari
2. mode: apakah menggunakan BFS atau DFS
3. amount: jumlah recipe yang ingin dicari

Berikut contoh request GET untuk mendapatkan satu recipe untuk elemen “Brick” menggunakan metode BFS.

```
http://localhost:8080/api/tree?element=Brick&mode=bfs&amount=1
```

JSON yang dihasilkan akan seperti berikut.

```

{
  "algorithm": "bfs",
  "depth": 2,
  "node_count": 5,
  "recipe_count": 1,
  "time": 18,
  "tree_data": {
    "element": "Brick",
    "children": [
      {
        "item_1": {
          "element": "Mud",
          "children": [
            {
              "item_1": {
                "element": "Water",
                "children": []
              },
              "item_2": {
                "element": "Earth",
                "children": []
              }
            }
          ]
        },
        "item_2": {
          "element": "Fire",
          "children": []
        }
      }
    ]
  }
}

```

Berikut konstruktor utamanya untuk melakukan ini.

prosedur **NewRouter**

```

procedure NewRouter(repository):

  create default router

  enable CORS with settings:
  - allow requests from http://localhost:3000
  - allow GET, POST, OPTIONS methods
  - allow Origin and Content-Type headers
  - expose Content-Length header
  - allow credentials

  define GET route at "/api/tree":

    get "element" from query
    if element is empty:
      respond with error "element parameter is required"
      stop

    get "mode" from query (default to "BFS")
    convert mode string to traversal mode

    get "amount" from query (default to "1")
    convert to integer, if invalid or ≤ 0, set amount to 1

    create a new tree builder using the repository and traversal mode
    if builder creation fails:
      respond with internal server error

```

```

    stop

    build the recipe tree using element and amount
    if build fails:
        respond with internal server error
        stop

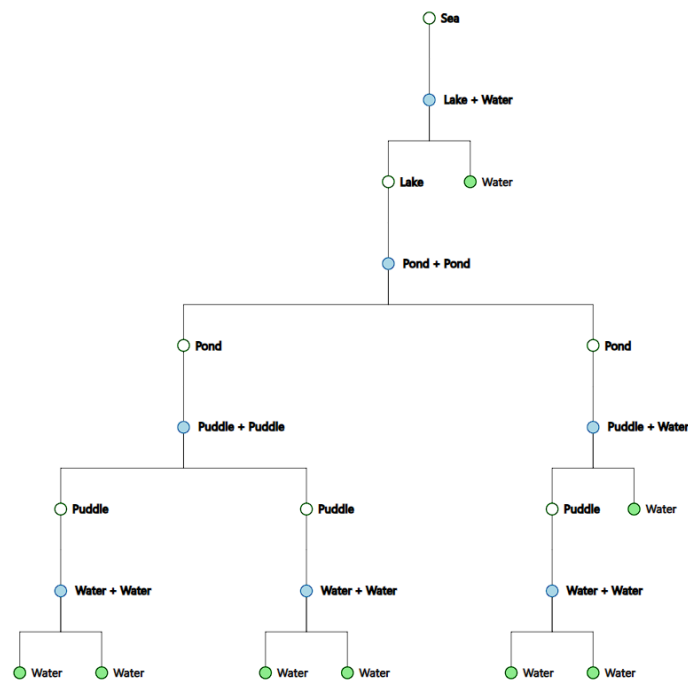
    respond with the tree as JSON

    return the configured router

```

4.2.3. Struktur Pohon Dinamis

Struktur dari pohon ruang status yang akan dibangun secara dinamis adalah RecipeTree, dengan adanya dua jenis simpul: ElementNode dan RecipeNode. Sebagai contoh, lihat gambar berikut yaitu pohon recipe dari Sea.



Gambar

Simpul yang merepresentasikan suatu elemen adalah ElementNode, yang diwarnakan putih atau hijau (jika simpul daun) di gambar, sedangkan simpul yang merepresentasikan penggabungan dua elemen adalah RecipeNode, yang diwarnakan biru. Berikut atribut dari simpul ElementNode.

Atribut	Jenis	Deskripsi
Element	string	Nama elemen dari simpul.
Ingredients	[]RecipeNode	Simpul recipe untuk elemen.
ParentRecipe	RecipeNode	Parent simpul recipe dari elemen.

IsPrimary	boolean	Apakah elemen primer.
Depth	int	Kedalaman simpul pada pohon.
RecipeCount	uint64	Jumlah recipe di bawah elemen ini pada pohon.

Berikut atribut dari simpul RecipeNode.

Atribut	Jenis	Deskripsi
Item1	RecipeNode	Unsur pertama pembangunan recipe.
Item2	RecipeNode	Unsur kedua pembangunan recipe.
ParentElement	[]RecipeNode	Parent simpul elemen dari resep.
Depth	int	Kedalaman simpul elemen anak pada pohon.
RecipeCount	uint64	Jumlah recipe di bawah elemen ini pada pohon.

Perhatikan berdasarkan konstruksi struktur simpul bahwa pohon akan direpresentasikan menggunakan sesuatu yang mirip “*doubly linked list*”, supaya dapat mengetahui kedua parent dan child dari suatu simpul. Berikut adalah atribut dari simpul RecipeTree.

Atribut	Jenis	Deskripsi
Mode	string	Metode penelusuran: DFS atau BFS.
Depth	int	Kedalaman pohon, yang dihitung berdasarkan <i>simpul elemen saja</i> .
NodeCount	[]RecipeNode	Jumlah simpul yang <i>ditelusuri</i> saat pencarian (bukan jumlah simpul pada pohon di akhir setelah simpul mati dihilangkan).
RecipeCount	uint64	Jumlah recipe yang direpresentasikan pohon.
Time	int	Waktu yang dibutuhkan untuk membangun pohon ini.
Root	RecipeNode	

Perhatikan bahwa walaupun ada dua jenis simpul, **hanya ElementNode yang dianggap sebagai simpul nyata**. Pertama, ini berarti jumlah simpul dari pohon adalah jumlah ElementNode yang **ditelusuri** (dijelajahi), sehingga simpul mati (yang disingkirkan di akhir sebelum divisualisasikan) termasuk. Kedua, ini berarti kedalaman pohon dihitung dari kedalaman

ElementNode terendah relatif ElementNode lainnya (sehingga *kedalaman suatu pohon recipe akan sama dengan tier dari elemen akarnya*, yaitu konsekuensi dari pengambilan recipe yang hanya mengandung unsur dengan tier lebih rendah).

Selanjutnya, pohon RecipeTree ini memiliki banyak metode maupun fungsi dan prosedur pembantu dari pohon (metode memiliki input RecipeTree).

Metode	Input	Output	Deskripsi
SetRecipeCount	tree RecipeTree	-	Menyimpan jumlah recipe akar ke pohonnya.
CountRecipes	tree RecipeTree node ElementNode	uint64	Menghitung ulang recipe dari pohon dengan mem- <i>bubble</i> informasi jumlah recipe dari simpul input ke atas branch-nya.
bubbleRecipes	node RecipeNode	-	Fungsi rekursif yang melakukan <i>bubble</i> jumlah recipe mulai dari simpul recipe.
bubbleNodes	node ElementNode	-	Fungsi rekursif yang melakukan <i>bubble</i> jumlah recipe mulai dari simpul elemen.
PruneTree	tree RecipeTree	-	Melakukan <i>pruning</i> simpul mati, yaitu simpul yang walaupun ditelusuri, tidak berkontribusi pada jumlah recipe pohon.
pruneNode	node ElementNode	-	Melakukan <i>pruning</i> simpul mati mulai dari simpul input.
TrimTree	tree RecipeTree amount int	-	Melakukan <i>pruning</i> simpul sehingga jumlah recipe pada pohon sudah sesuai (atau setidaknya mencoba) dengan yang diinginkan.
getLeafRecipes	node ElementNode	[]RecipeNode	Mengembalikan semua RecipeNode daun, yaitu simpul recipe yang hanya memiliki anak simpul elemen primer.
String	tree RecipeTree	string	Menghasilkan string visualisasi pohon yang dihasilkan dengan informasi debugging khusus

			(seperti jumlah recipe pada setiap simpul, waktu pencarian, kedalaman, dll.).
printNode	<pre> tree RecipeTree sb strings.Builder node ElementNode indentLevel int </pre>	-	Fungsi rekursif untuk membangun string dari data pohonnya sendiri, dengan indenting yang benar, supaya terlihat di <i>console</i> dengan baik.

Ada beberapa catatan penting lainnya mengenai metode pohon yang diimplementasikan ini.

1. CountRecipes akan digunakan untuk menghitung jumlah recipe dengan memanggilkannya saat suatu simpul terbentuk (atau jika ingin lebih efisien, pada simpul daun). Dia menggunakan konsep *bubbling*, yaitu dengan menghantarkan informasi jumlah recipe pada suatu simpul menuju atas pohon pada *branch*-nya (seperti xilem pada pohon makhluk hidup sebenarnya) akibat perubahan (seperti penambahan simpul) di simpul tersebut. Untuk melakukan itu, digunakan prosedur bubbleRecipes dan bubbleNodes mulai dari simpul tersebut.

fungsi CountRecipes
Input: tree RecipeTree , node ElementNode Output: recipeCount int
<pre> function CountRecipes: call bubbleNodes on input node call SetRecipeCount on tree return RecipeCount of tree </pre>
prosedur BubbleRecipes
<pre> procedure bubbleRecipes: if node is null: stop set node's recipe count to: Item1's recipe count × Item2's recipe count if node has a parent element: call bubbleNodes on the parent element </pre>
prosedur bubbleNodes
<pre> procedure bubbleNodes: if node is null: stop initialize count to 0 for each recipe in node's ingredients: add recipe's recipe count to count set node's recipe count to the total count </pre>


```
if node has a parent recipe:  
    call bubbleRecipes on the parent recipe
```

2. PruneTree akan menghilangkan semua simpul mati yang dihasilkan pohon dinamis ini (hanya masalah untuk BFS sebenarnya). Walaupun dilakukan *pruning* simpul mati, **jumlah simpul tidak diubah**, karena diasumsikan informasi jumlah simpul (*node count*) dari pohon adalah jumlah simpul yang ditelusuri atau dijelajahi (termasuk simpul mati).

prosedur **PruneTree**

Input: tree **RecipeTree**

Output: -

```
procedure PruneTree:  
    call pruneNode on tree root
```

prosedur **pruneNode**

```
procedure pruneNode:  
    if elementNode is null:  
        stop  
  
    create an empty list called trimmed  
  
    for each recipe in node's ingredients:  
        if the recipe is null or its recipe count is 0:  
            skip it  
  
        call pruneNode on recipe's Item1  
        call pruneNode on recipe's Item2  
  
        add the recipe to trimmed list  
  
    replace node's ingredients with the trimmed list
```

3. TrimTree dibutuhkan untuk memotong beberapa simpul yang membuat jumlah recipe pada pohon terlalu banyak. Ini diperlukan karena kami menggunakan *multithreading* pada pembuatan pohon (jika tidak, pohon yang dihasilkan sudah pasti akan memiliki jumlah recipe yang sesuai, sehingga ini tidak diperlukan). Saat suatu thread telah selesai melakukan suatu ekspansi node yang mengakibatkan jumlah recipe sudah sesuai, informasi penghentian tidak bisa secara langsung diberikan ke thread lainnya karena mereka sedang bekerja secara paralel ekspansi node mereka sendiri, dan hanya dapat mengetahui untuk berhenti setelah mereka selesai. Ini dapat menghasilkan kelebihan simpul, sehingga TrimTree dibutuhkan.

procedure TrimTree

Input: tree **RecipeTree**, amount **int**

Output: -

```
procedure TrimTree:  
    if tree's recipe count is less than or equal to amount:
```

<pre> stop get leaf recipes from the root of the tree sort the leaves by depth, deepest first for each leaf in the sorted list: if tree's recipe count equals the desired amount: stop temporarily set the leaf's recipe count to 0 update recipe count starting from the leaf's parent element if tree's recipe count is now too low: restore the leaf's recipe count using bubbleRecipes recalculate the total recipe count </pre>
fungsi getLeafRecipes
Input: node ElementNode Output: list of RecipeNode
<pre> function getLeafRecipes: initialize an empty list of leaves for each recipe in node's ingredients: if recipe is valid and both children are primary elements: add recipe to leaves list else: get leaf recipes from Item1 and add to list of leaves get leaf recipes from Item2 and add to list of leaves return the list of leaf recipes </pre>

4.2.4. Builder

Untuk membuat kode lebih modular, dibuat *interface* Builder. Ini adalah “benda pembangun” yang akan melakukan pembuatan pohon dinamis dengan metode BuildTree-nya. Metode tersebut bisa BFS atau DFS, sehingga Builder bisa menjadi pembangun BFS atau DFS sesuai input dari request HTTP. Konstruktor untuk *interface* dari Builder sendiri seperti berikut.

konstruktor Builder
Input: repository RecipeRepository , traversalMode string Output: a Builder structure
<pre> constructor NewBuilder: if traversalMode is BFS: return a new BFSBuilder using the repository else if traversalMode is DFS: return a new DFSBuilder using the repository else: return an error saying "invalid mode" </pre>

Lalu, fungsi BuildTree yang dimaksud seperti berikut.

Metode	Input	Output	Deskripsi
BuildTree	rootElement string amount int	RecipeTree	Membangun pohon recipe secara dinamis menggunakan BFS atau DFS, tergantung Builder yang diinisialisasi.

4.2.5. BFS

kami membuat BFSBuilder sebagai suatu jenis Builder yang mengimplementasi pencarian *Breadth First Search* (BFS). Sesuai *interface* Builder, fungsi BuildTree-nya yang melakukan penelusuran secara BFS tersebut. Berikut *source code*-nya secara langsung agar terlihat jelas BFS yang diimplementasikan.

fungsi BuildTree
Input: rootElement string , amount int Output: *model.RecipeTree, error
<pre> start := time.Now() var (wg sync.WaitGroup mu sync.Mutex) ctx, cancel := context.WithCancel(context.Background()) defer cancel() tree := model.NewTree(rootElement, model.BFS) queue := model.NewQueue(tree.Root) for !queue.IsEmpty() { if ctx.Err() != nil tree.RecipeCount >= uint64(amount) { break } current := queue.Pop() if current.IsPrimary { continue } recipes, err := b.repo.GetRecipesFor(current.Element) if err != nil { cancel() wg.Wait() return nil, err } for _, recipe := range recipes { wg.Add(1) go func(recipe *model.Recipe) { defer wg.Done() if ctx.Err() != nil { return } </pre>

```

    }

    item1 := model.NewElementNode(recipe.Item1, nil, current.Depth + 1)
    item2 := model.NewElementNode(recipe.Item2, nil, current.Depth + 1)

    recipeNode := model.NewRecipeNode(item1, item2, current)

    item1.ParentRecipe = recipeNode
    item2.ParentRecipe = recipeNode

    mu.Lock()
    defer mu.Unlock()

    current.Ingredients = append(current.Ingredients, recipeNode)
    tree.NodeCount += 2

    if item1.Depth > tree.Depth {
        tree.Depth = item1.Depth
    }

    if item1.IsPrimary || item2.IsPrimary {
        tree.CountRecipes(current)
    }

    if tree.RecipeCount >= uint64(amount) {
        cancel()
        return
    }

    queue.Push(item1)
    queue.Push(item2)

}(recipe)
}
wg.Wait()
}

tree.TrimTree(amount)
tree.PruneTree()

tree.Time = int(time.Since(start).Milliseconds())

return tree, nil

```

Ada beberapa hal yang perlu dikomentari mengenai implementasi ini.

1. Pertama, kami menggunakan antrian atau *queue* untuk mengatur urutan penjelajahan simpul di BFS. Data struktur antrian tersebut dibuat sendiri, dan dibuat kompatibel dengan *multithreading* menggunakan mutex. Adapun bahwa antrian mendukung beberapa metode antrian seumumnya, seperti Pop, Push, dan IsEmpty.

Implementasi Antrian: Queue

```

type Queue [T any] struct {
    items []T
    mu     sync.Mutex
}

func NewQueue[T any](initial ...T) *Queue[T] {
    return &Queue[T]{
        items: initial,
    }
}

```

```

}

func (q *Queue[T]) Push(x T) {
    q.mu.Lock()
    defer q.mu.Unlock()
    q.items = append(q.items, x)
}

func (q *Queue[T]) Pop() (T) {
    q.mu.Lock()
    defer q.mu.Unlock()

    var zero T
    if len(q.items) == 0 {
        return zero
    }

    item := q.items[0]
    q.items = q.items[1:]
    return item
}

func (q *Queue[T]) IsEmpty() bool {
    q.mu.Lock()
    defer q.mu.Unlock()
    return len(q.items) == 0
}

```

2. Seperti implementasi BFS umumnya, ditelusuri setiap simpul pada antrian (yang awalnya adalah akar dari pohon, yaitu elemen yang ingin dicari resepnya) sampai dia kosong, atau untuk pohon dinamis, itu sampai ditemukan solusi. Dalam kasus ini, solusi berupa perolehan jumlah recipe unik yang sesuai *request*.
3. Di setiap iterasi pembacaan simpul di antrian, kami akan mencari semua recipe dari elemen pada simpul menggunakan repository. Lalu, akan dibuat suatu simpul recipe serta dua simpul elemen untuk setiap resep. Simpul elemen akan ditambah ke antriannya lagi sedangkan simpul recipe akan ditambah sebagai suatu recipe (anak simpul) dari elemen yang sedang ditinjau. Lalu, informasi seperti jumlah simpul dan kedalaman pohon di-*update*.
4. Simpul yang berupa elemen primer tidak akan diproses, karena ini akan menjadi simpul daun pohon.
5. Untuk mengetahui berapa recipe yang telah ada di pohon, kami panggil fungsi CountRecipes. Tetapi, karena teknis kerjanya, sebenarnya kami bisa memanggilnya *hanya pada simpul pseudo-daun*, atau simpul yang anaknya adalah simpul daun. Mengapa? Karena *bubbling* jumlah recipe yang dilakukan akan berulang atau mubazir pada simpul-simpul pada *branch* yang sama pada pohon. Jadi, cukup melakukannya di kedalaman $n - 1$ pohon.
6. Diimplementasikan *multithreading* pada pembentukan recipe baru untuk suatu elemen. Jadi, untuk suatu elemen dengan banyak recipe, dapat dilakukan secara konkuren maupun paralel. Ini dilakukan dengan meluncurkan goroutine untuk setiap recipe, lalu kami menggunakan WaitGroups untuk menunggu semua goroutine untuk selesai.

Adapun penggunaan mutex saat melakukan perubahan pada pohon supaya tidak terjadi interferensi antar goroutine.

7. Saat jumlah recipe pada pohon sudah sama atau melebihi yang diinginkan, pencarian harus dihentikan. Ini dilakukan dengan *me-return*-kan fungsinya. Tetapi, diimplementasikan juga penggunaan package *context*, yaitu untuk mengkomunikasikan sinyal penghentian pada semua goroutine, sehingga dapat determinasikan dengan efisien.

4.2.6. DFS

Mirip dengan BFS, kami membuat DFSBuilder sebagai suatu jenis Builder yang mengimplementasi pencarian *Depth First Search* (DFS). Sesuai *interface* Builder, fungsi BuildTree-nya yang melakukan penelusuran secara DFS tersebut. Berikut *source code*-nya secara langsung agar terlihat jelas DFS yang diimplementasikan.

fungsi BuildTree

Input: rootElement **string**, amount **int**

Output: *model.RecipeTree, error

```
start := time.Now()

// Multithreading tools
var (
    wg sync.WaitGroup
    mu sync.Mutex
)
ctx, cancel := context.WithCancel(context.Background())
defer cancel()

tree := model.NewTree(rootElement, model.DFS) // Start tree
stack := model.NewStack(tree.Root)           // Add root to queue

// Loop through queue
for !stack.IsEmpty() {

    // Stop if recipe count exceeded
    if ctx.Err() != nil || tree.RecipeCount >= uint64(amount) {
        break
    }

    // Pop node from stack
    current := stack.Pop()

    // Skip primary element
    if current.IsPrimary {
        continue
    }

    // Get recipes for current element
    recipes, err := d.repo.GetRecipesFor(current.Element)
    if err != nil {
        cancel()
        wg.Wait()
        return nil, err
    }

    // Make new recipe node for each recipe
    for _, recipe := range recipes {
```

```

        wg.Add(1)

        go func(recipe *model.Recipe) {
            defer wg.Done()

            if ctx.Err() != nil {
                return
            }

            // New element node
            item1 := model.NewElementNode(recipe.Item1, nil, current.Depth + 1)
            item2 := model.NewElementNode(recipe.Item2, nil, current.Depth + 1)

            // New recipe node
            recipeNode := model.NewRecipeNode(item1, item2, current)
            item1.ParentRecipe = recipeNode
            item2.ParentRecipe = recipeNode

            mu.Lock()
            defer mu.Unlock()

            // New ingredient
            current.Ingredients = append(current.Ingredients, recipeNode)

            // Update node count
            tree.NodeCount += 2

            // Update depth
            if item1.Depth > tree.Depth {
                tree.Depth = item1.Depth
            }

            // Recount recipes
            if item1.IsPrimary && item2.IsPrimary {
                tree.CountRecipes(current)
            }

            // Stop if found enough recipes
            if tree.RecipeCount >= uint64(amount) {
                cancel()
                return
            }

            stack.Push(item1)
            stack.Push(item2)

        }(recipe)
    }

    wg.Wait()
}

// Polish tree
tree.TrimTree(amount)
tree.PruneTree()

// Time tree
elapsed := time.Since(start)
tree.Time = int(elapsed.Milliseconds())

return tree, nil

```

Terlihat bahwa sebenarnya implementasi DFS mirip sekali dengan implementasi BFS. Digunakan pendekatan tumpukan (*stack*), yang menggantikan antrian pada BFS. Alasan kami menggunakan tumpukan daripada rekursi untuk mengimplementasi DFS adalah untuk

memudahkan pengembangannya (karena kodenya mirip untuk BFS dan DFS kalau seperti itu), tetapi juga agar *multithreading* menggunakan goroutines akan lebih jelas, sebab implementasi goroutines dalam rekursi sangat membelit dan sulit. Untuk lebih jelasnya, berikut data struktur tumpukan (*stack*) yang kami gunakan.

Implementasi Tumpukan: Stack
<pre> type Stack[T any] struct { items []T mu sync.Mutex } func NewStack[T any](initial ...T) *Stack[T] { return &Stack[T]{ items: initial, } } func (s *Stack[T]) Push(x T) { s.mu.Lock() defer s.mu.Unlock() s.items = append(s.items, x) } func (s *Stack[T]) Pop() T { s.mu.Lock() defer s.mu.Unlock() var zero T if len(s.items) == 0 { return zero } i := len(s.items) - 1 item := s.items[i] s.items = s.items[:i] return item } func (s *Stack[T]) IsEmpty() bool { s.mu.Lock() defer s.mu.Unlock() return len(s.items) == 0 } </pre>

Selain itu, logika pembacaan simpul dari tumpukan, pemrosesan simpul daun, perhitungan jumlah recipe, dan implementasi *multithreading* sama persis dengan BFS. Hanya saja, karena penggunaan tumpukan daripada antrian, pembentukan pohon dinamis tidak lagi per level, melainkan mendalam per *branch*.

4.3. Implementasi Frontend

Frontend dari aplikasi web menggunakan bahasa Javascript dengan *framework* React.js. Struktur direktori frontend adalah sebagai berikut:

Struktur Direktori Frontend

```

frontend/
├── Dockerfile*
├── nginx.conf*
├── package-lock.json*
├── package.json*
├── public/
│   └── index.html*
└── src/
    ├── App.js*
    ├── components/
    │   ├── RecipeResults/
    │   │   ├── RecipeResults.css*
    │   │   └── RecipeResults.js*
    │   ├── RecipeTree/
    │   │   ├── RecipeTree.css*
    │   │   └── RecipeTree.js*
    │   ├── ResetButton/
    │   │   ├── ResetButton.css*
    │   │   └── ResetButton.js*
    │   ├── StatsPanel/
    │   │   ├── StatsPanel.css*
    │   │   └── StatsPanel.js*
    │   └── SearchControls/
    │       ├── AlgorithmToggle/
    │       │   ├── AlgorithmToggle.css*
    │       │   └── AlgorithmToggle.js*
    │       ├── MaxPathsInput/
    │       │   ├── MaxPathsInput.css*
    │       │   └── MaxPathsInput.js*
    │       ├── ModeToggle/
    │       │   ├── ModeToggle.css*
    │       │   └── ModeToggle.js*
    │       ├── SearchBar/
    │       │   ├── SearchBar.css*
    │       │   └── SearchBar.js*
    │       ├── SearchControls.css*
    │       └── SearchControls.js*
    ├── index.js*
    ├── pages/
    │   └── HomePage.js*
    ├── reportWebVitals.js*
    ├── setupTests.js*
    ├── styles/
    │   ├── HomePage.css*
    │   └── index.css*
    └── utils/
        ├── api.js*
        ├── contoh1.json*
        ├── contoh2.json*
        └── formatTree.js*

```

4.3.1. Components

Komponen-komponen *frontend* merupakan fungsi-fungsi dalam JavaScript yang menghasilkan JSX (JavaScript XML) yang nantinya akan di-*render* menjadi komponen dalam web.

```
frontend/src/components/SearchControls/
```

```

import React from 'react';
import './AlgorithmToggle.css';
import './MaxPathsInput.css';

```

```

import './ModeToggle.css';
import './SearchBar.css';
import './SearchControls.css';

export function AlgorithmToggle({ algorithm, onChange }) {
  return (
    <div className="algorithm-toggle">
      <button
        className={`toggle-button ${algorithm === 'bfs' ? 'active' : ''}`}
        onClick={() => onChange('bfs')}
      >
        BFS
      </button>
      <button
        className={`toggle-button ${algorithm === 'dfs' ? 'active' : ''}`}
        onClick={() => onChange('dfs')}
      >
        DFS
      </button>
    </div>
  );
};

export function MaxPathsInput({ value, onChange }) {
  return (
    <div className="max-paths-input">
      <label htmlFor="maxPaths">Max Paths:</label>
      <input
        type="number"
        id="maxPaths"
        min="2"
        max="100"
        value={value}
        onChange={(e) => onChange(e.target.value)}
      />
    </div>
  );
};

export function ModeToggle({ mode, onChange }) {
  return (
    <div className="mode-toggle">
      <button
        className={`toggle-button ${mode === 'single' ? 'active' : ''}`}
        onClick={() => onChange('single')}
      >
        Single Path
      </button>
      <button
        className={`toggle-button ${mode === 'multiple' ? 'active' : ''}`}
        onClick={() => onChange('multiple')}
      >
        Multi Paths
      </button>
    </div>
  );
};

export function SearchBar({ value, onChange, onSubmit }) {
  function handleSubmit(e) {
    e.preventDefault(); // Prevent default form submission
    onSubmit(e);         // Teruskan event ke parent
  };

  return (
    <form className="search-bar" onSubmit={handleSubmit}> {/* Gunakan handler lokal */}
      <input
        type="text"

```

```

        value={value}
        onChange={onChange}
        placeholder="Search element (e.g. Brick, Water)..."
        className="search-input"
      />
      <button type="submit" className="search-button">
        Search
      </button>
    </form>
  );
};

export function SearchControls({
  searchTerm,
  onSearchChange,
  onSearchSubmit,
  algorithm,
  onAlgorithmChange,
  mode,
  onModeChange,
  maxPaths,
  onMaxPathsChange
}) {
  return (
    <div className="search-controls">
      <SearchBar
        value={searchTerm}
        onChange={onSearchChange}
        onSubmit={onSearchSubmit}
      />
      <div className="search-options">
        <AlgorithmToggle
          algorithm={algorithm}
          onChange={onAlgorithmChange}
        />
        <ModeToggle
          mode={mode}
          onChange={onModeChange}
        />
        {mode === 'multiple' && (
          <MaxPathsInput
            value={maxPaths}
            onChange={onMaxPathsChange}
          />
        )}
      </div>
    </div>
  );
};

```

frontend/src/components/RecipeResults/

```

import React from 'react';
import { Tree } from 'react-d3-tree';
import { formatTree } from '../../../utils/formatTree';
import './RecipeTree.css';
import './ResetButton.css';
import './StatsPanel.css';
import './RecipeResults.css';

// Render custom
function renderCustomNode({ nodeDatum }) {
  const isRecipe = nodeDatum.attributes?.type === 'recipe';
  const isBase = nodeDatum.attributes?.type === 'base-element';

  return (

```

```

    <g>
      <circle
        r="10"
        fill={isRecipe ? 'lightblue' : isBase ? 'lightgreen' : 'white'}
        stroke={isRecipe ? 'steelblue' : 'darkgreen'}
        strokeWidth="2"
      />
      <text
        x={isRecipe ? 25 : 20}
        dy={isRecipe ? '.31em' : '.35em'}
        fill={isRecipe ? 'navy' : 'black'}
        fontSize={isRecipe ? '20px' : '20px'}
      >
        {nodeDatum.name}
      </text>
    </g>
  );
};

export function RecipeTree({ data }) {
  return (
    <div className="recipe-tree-container">
      <Tree
        data={formatTree(data)}
        orientation="vertical"
        pathFunc="step"
        translate={{ x: 200, y: 50 }}
        renderCustomNodeElement={renderCustomNode}
        className="recipe-tree"
      />
    </div>
  );
};

export function ResetButton({ onClick }) {
  return (
    <button className="reset-button" onClick={onClick}>
      Reset
    </button>
  );
};

export function StatsPanel({ stats, mode }) {
  return (
    <div className="stats-panel">
      <h3>Search Statistics</h3>
      <div className="stats-tables">
        <table className="stats-table">
          <tbody>
            <tr>
              <td className="stat-label">Algorithm :</td>
              <td className="stat-value">{stats.algorithm.toUpperCase()}</td>
            </tr>
            <tr>
              <td className="stat-label">Depth :</td>
              <td className="stat-value">{stats.depth}</td>
            </tr>
            <tr>
              <td className="stat-label">Nodes :</td>
              <td className="stat-value">{stats.node_count}</td>
            </tr>
          </tbody>
        </table>

        <table className="stats-table">
          <tbody>
            <tr>
              <td className="stat-label">Mode :</td>

```

```

        <td className="stat-value">
          {mode === 'single' ? 'Single Path' : 'Multiple Paths'}
        </td>
      </tr>
      <tr>
        <td className="stat-label">Recipes :</td>
        <td className="stat-value">{stats.recipe_count}</td>
      </tr>
      <tr>
        <td className="stat-label">Time :</td>
        <td className="stat-value">{stats.time} ms</td>
      </tr>
    </tbody>
  </table>
</div>
</div>
);
};

export function RecipeResults({ results, mode, onReset }) {
  if (!results) return null;

  return (
    <div className="recipe-results">
      <div className="results-header">
        <StatsPanel
          stats={results}
          mode={mode}
        />
        <ResetButton
          onClick={onReset}
        />
      </div>
      <RecipeTree
        data={results.tree_data}
      />
    </div>
  );
};

```

4.3.2. HomePage

HomePage merupakan gabungan dari seluruh komponen, serta mengatur input pengguna dan penggunaan API.

frontend/src/pages/HomePage.js

```

import React, { useState } from 'react';
import SearchControls from '../components/SearchControls/SearchControls';
import RecipeResults from '../components/RecipeResults/RecipeResults';
import { fetchRecipe } from '../utils/api';
import '../styles/HomePage.css';

function HomePage() {
  // State management
  const [searchTerm, setSearchTerm] = useState('');
  const [algorithm, setAlgorithm] = useState('bfs');
  const [mode, setMode] = useState('single');
  const [modeSubmitted, setModeSubmitted] = useState('single');
  const [maxPaths, setMaxPaths] = useState(2);
  const [results, setResults] = useState(null);
  const [isLoading, setIsLoading] = useState(false);

```

```

// Handlers
function handleSearchSubmit(e) {
  e.preventDefault();
  if (!searchTerm.trim()) return;
  setModeSubmitted(mode);
  setIsLoading(true);
  fetchRecipe({
    element: searchTerm,
    algorithm,
    maxPaths: mode === 'multiple' ? maxPaths : 1
  })
  .then(setResults)
  .catch(console.error)
  .finally(() => setIsLoading(false));
};

function handleReset() {
  setSearchTerm('');
  setAlgorithm('bfs');
  setMode('single');
  setModeSubmitted('single');
  setMaxPaths(2);
  setResults(null);
  setIsLoading(false);
};

return (
  <div className="home-page">
    <h1>Little Alchemy Recipe Finder</h1>

    <SearchControls
      // SearchBar props
      searchTerm={searchTerm}
      onSearchChange={(e) => setSearchTerm(e.target.value)}
      onSearchSubmit={handleSearchSubmit}

      // AlgorithmToggle props
      algorithm={algorithm}
      onAlgorithmChange={setAlgorithm}

      // ModeToggle props
      mode={mode}
      onModeChange={setMode}

      // MaxPathsInput props
      maxPaths={maxPaths}
      onMaxPathsChange={(value) => setMaxPaths(Number(value))}
    />

    {isLoading ? (
      <div className="loading">Searching recipes...</div>
    ) : (
      results &&
      <RecipeResults
        results={results}
        mode={modeSubmitted}
        onReset={handleReset}
      />
    )}
  </div>
);
};

export default HomePage;

```

4.3.3. Utils

Utils merupakan fungsi-fungsi JavaScript tambahan non-React yang penting dan menunjang kinerja fungsi-fungsi React.

frontend/src/utils/api.js

```
export async function fetchRecipe({ element, algorithm, maxPaths }) {
  // Konversi parameter frontend ke format backend
  const backendParams = {
    element: element,
    mode: algorithm,    // 'algorithm' di frontend -> 'mode' di backend
    amount: maxPaths   // Konversi mode frontend ke amount
  };

  const params = new URLSearchParams(backendParams);

  try {
    const response = await fetch(
      `http://localhost:8080/api/tree?${params}`, {
        method: 'GET',
        headers: {
          'Content-Type': 'application/json'
        }
      }
    );
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    } else {
      const contentType = response.headers.get('content-type');
      if (!contentType?.includes('application/json')) {
        throw new TypeError("Response is not JSON");
      }
    }
    return await response.json();
  } catch (error) {
    alert(`
      - error: ${error}
      - element: ${backendParams.element}
      - mode: ${backendParams.mode}
      - amount: ${backendParams.amount}
    `);
    throw error;
  }
}
```

frontend/src/utils/formatTree.js

```
// Helper function untuk membuat formatted tree agar mudah divisualisasikan di RecipeTree.js

export function formatTree(rawData) {
  function transformNode(node) {
    if (!node.children || node.children.length === 0) {
      return {
        name: node.element,
        attributes: {
          type: 'base-element'
        }
      };
    }
  };
}

// Proses setiap resep (pasangan item)
```

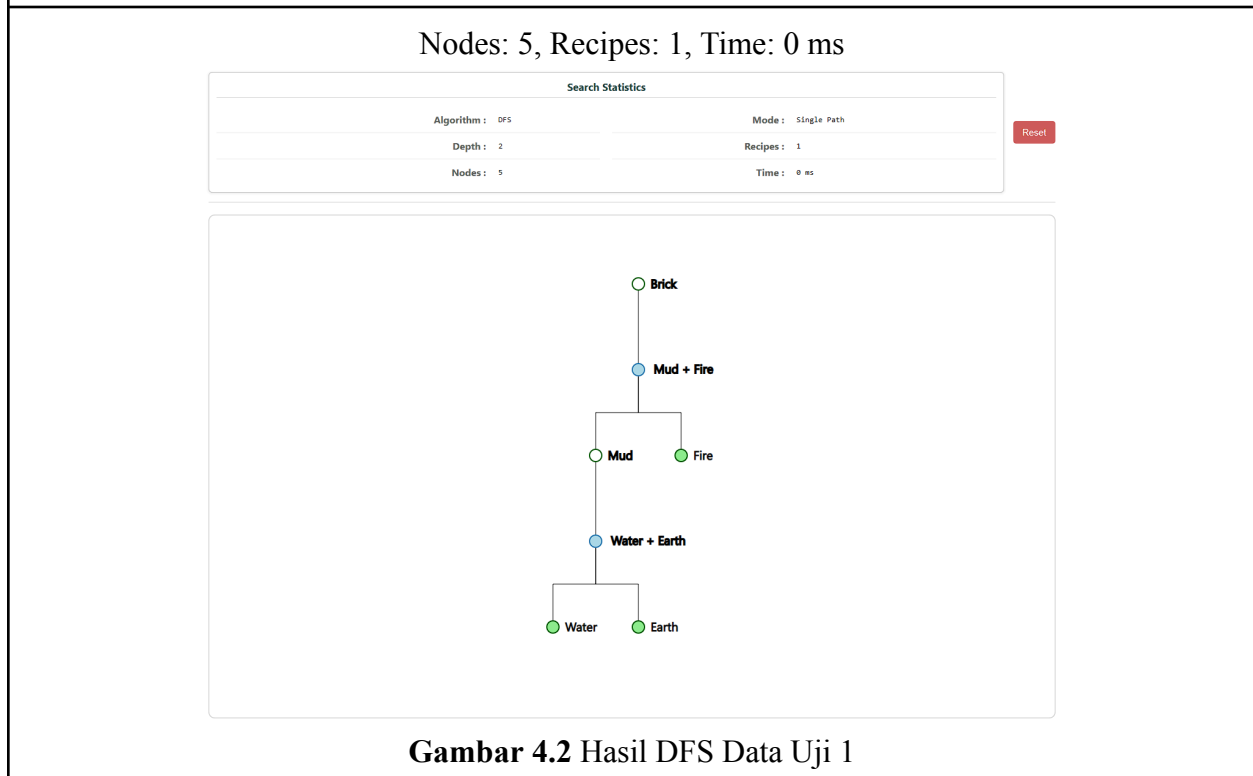
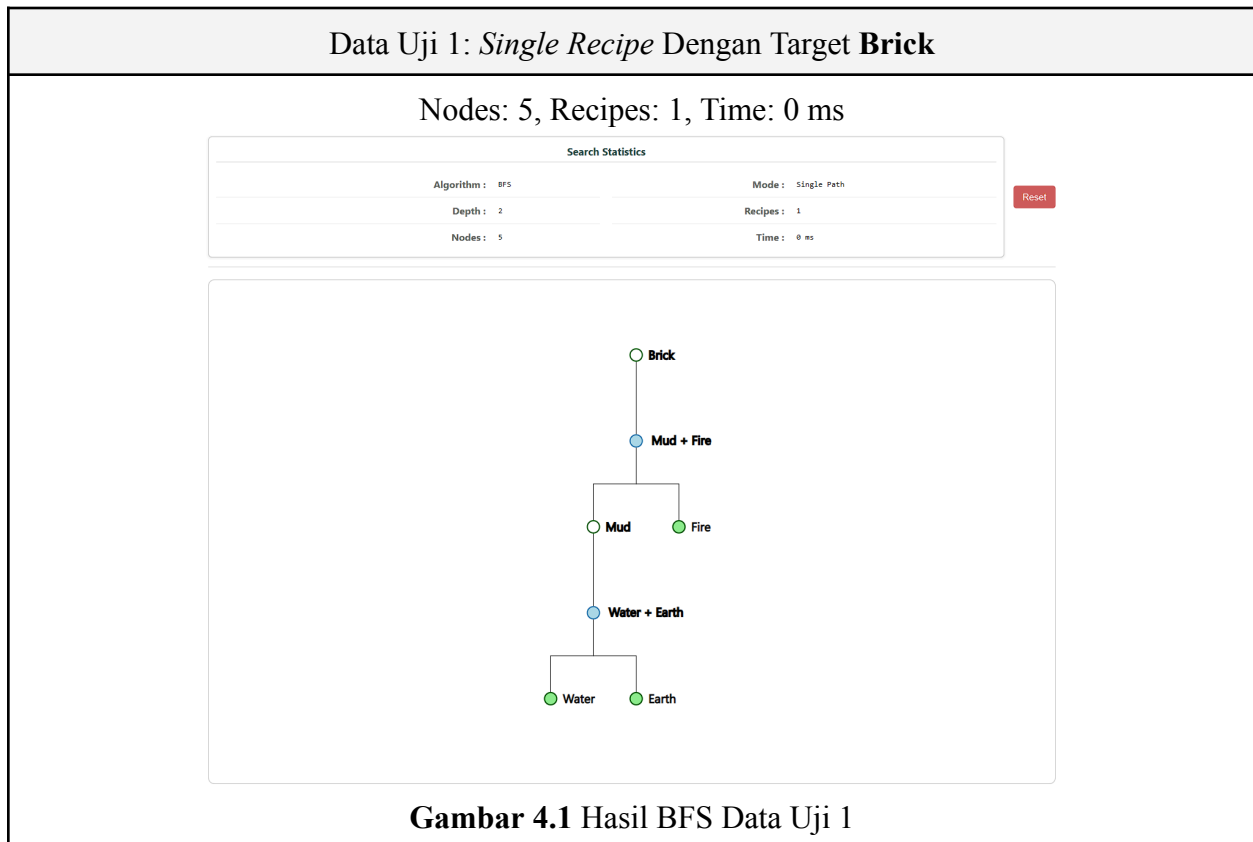
```

    return {
      name: node.element,
      children: node.children.map(recipe => ({
        name: `${recipe.item_1.element} + ${recipe.item_2.element}`,
        attributes: {
          type: 'recipe'
        },
        children: [
          transformNode(recipe.item_1),
          transformNode(recipe.item_2)
        ]
      })))
    };
  };

  return transformNode(rawData);
};

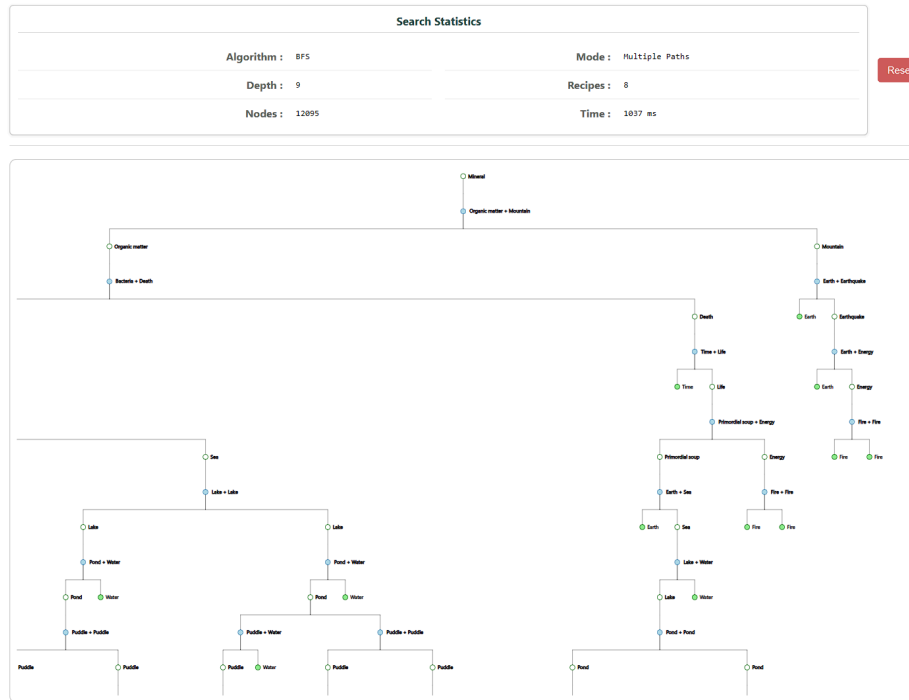
```


4.3. Pengujian



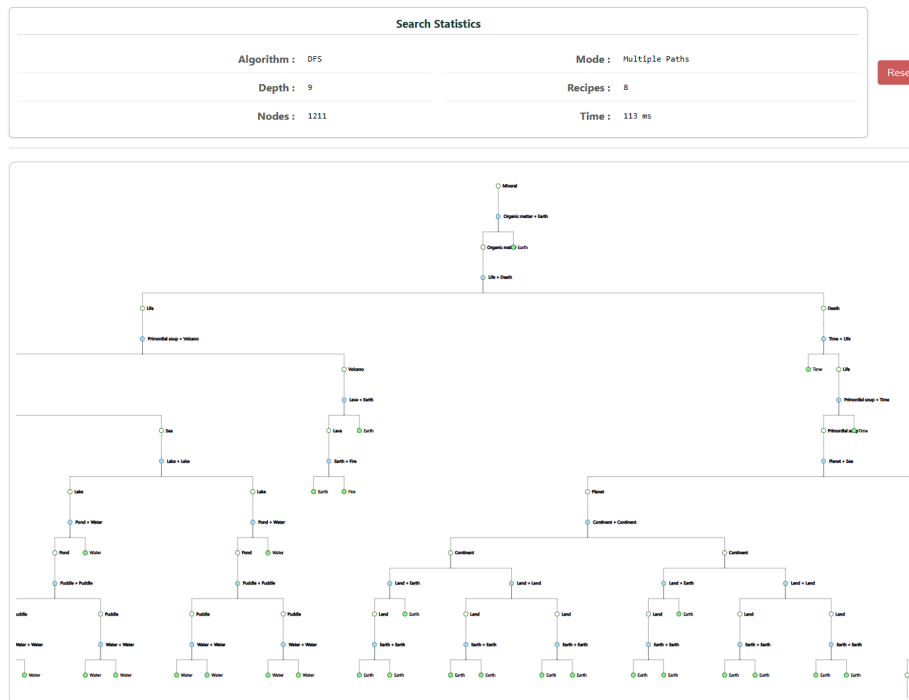
Data Uji 3: *Multiple Recipe* (8) Dengan Target **Mineral**

Nodes: 12095, Recipes: 8, Time: 1037 ms



Gambar 4.5 Hasil BFS Data Uji 3

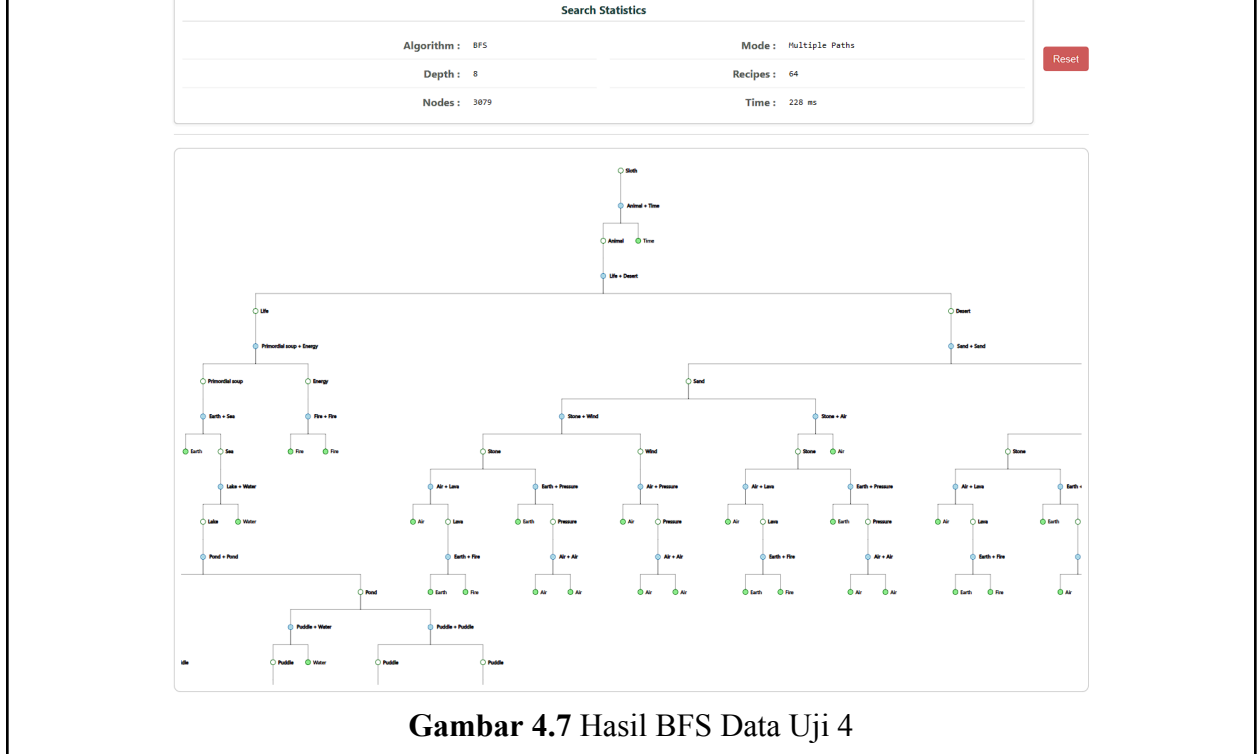
Nodes: 1211, Recipes: 8, Time: 113 ms



Gambar 4.6 Hasil DFS Data Uji 3

Data Uji 4: <i>Multiple Recipe</i> (64) Dengan Target Sloth
--

Nodes: 3079, Recipes: 64, Time: 228 ms

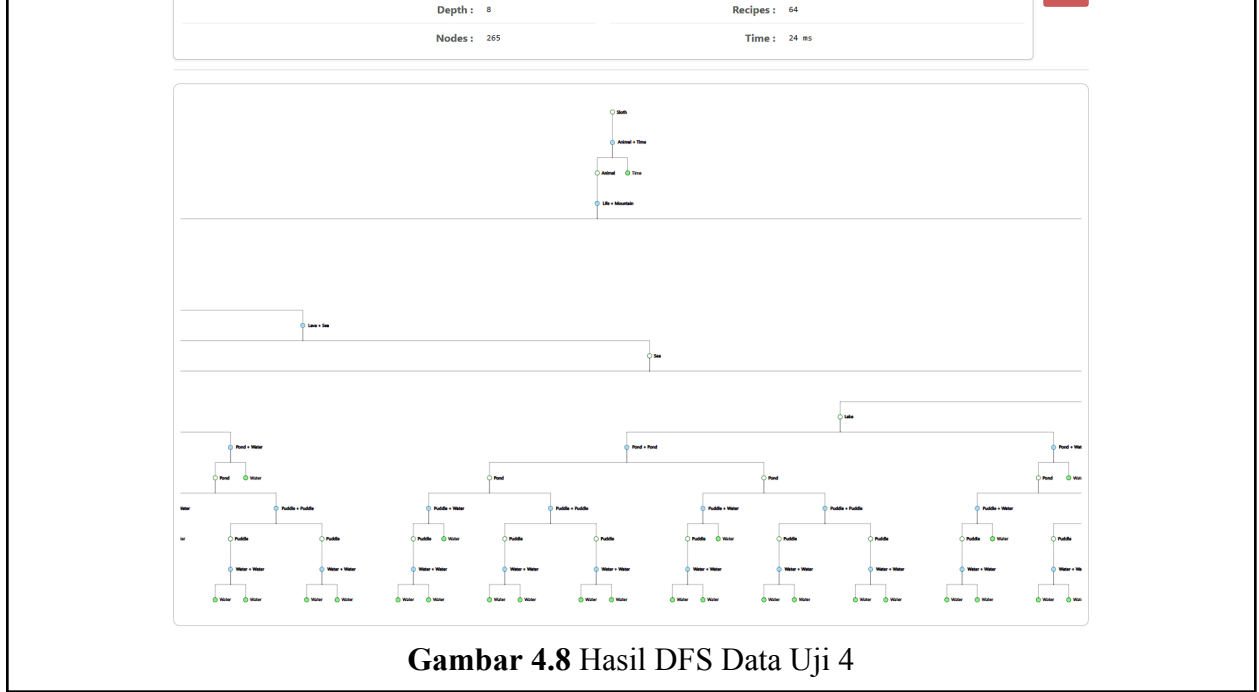


Nodes: 265, Recipes: 64, Time: 24 ms

Search Statistics

Algorithm : DFSMode : Multiple Paths

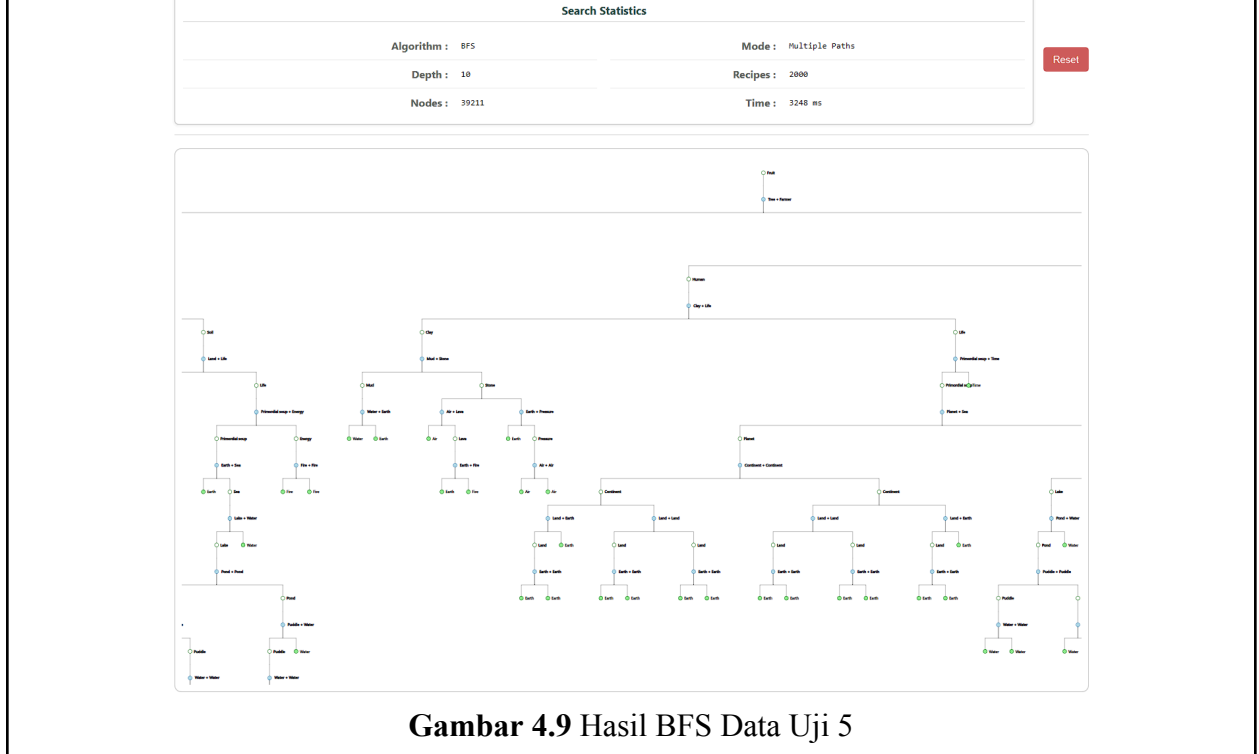
Reset



IF221 Strategi Algoritma
44

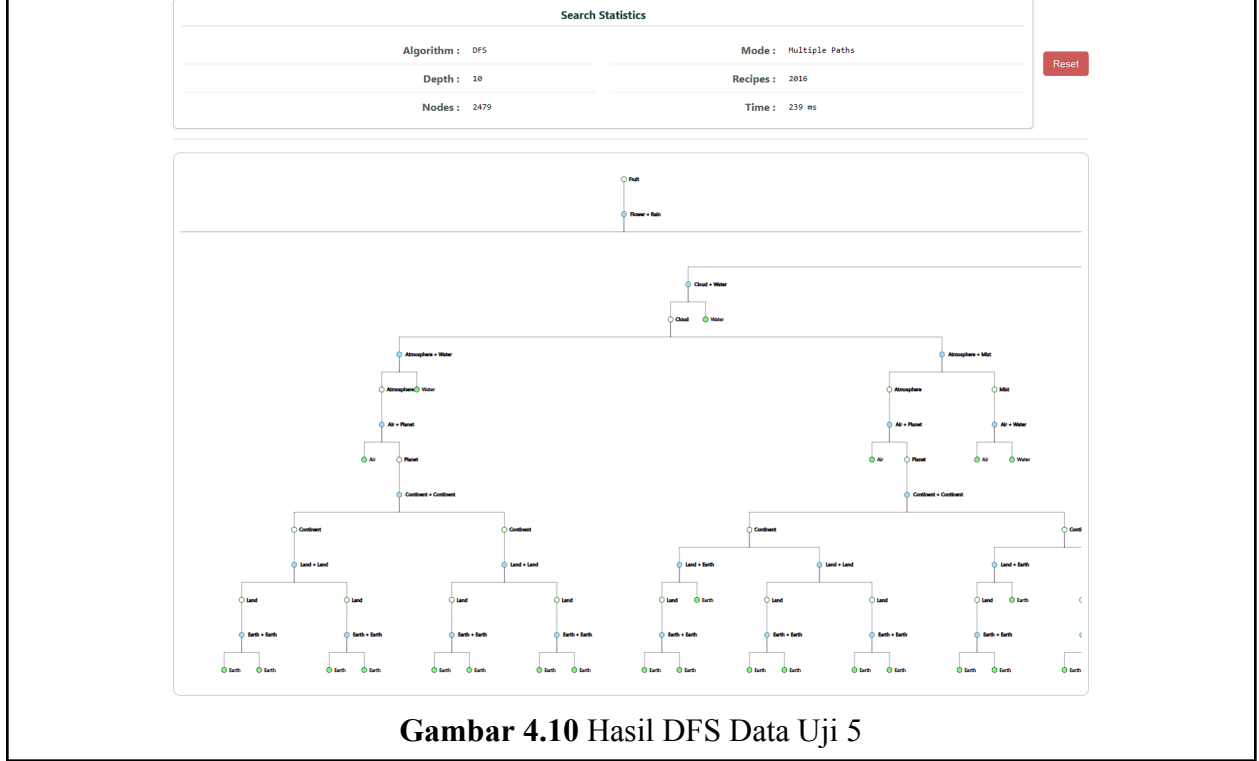
Data Uji 5: <i>Multiple Recipe</i> (2000) Dengan Target Fruit
--

Nodes: 39211, Recipes: 2000, Time: 3248 ms



Nodes: 2479, Recipes: 2016, Time: 239 ms

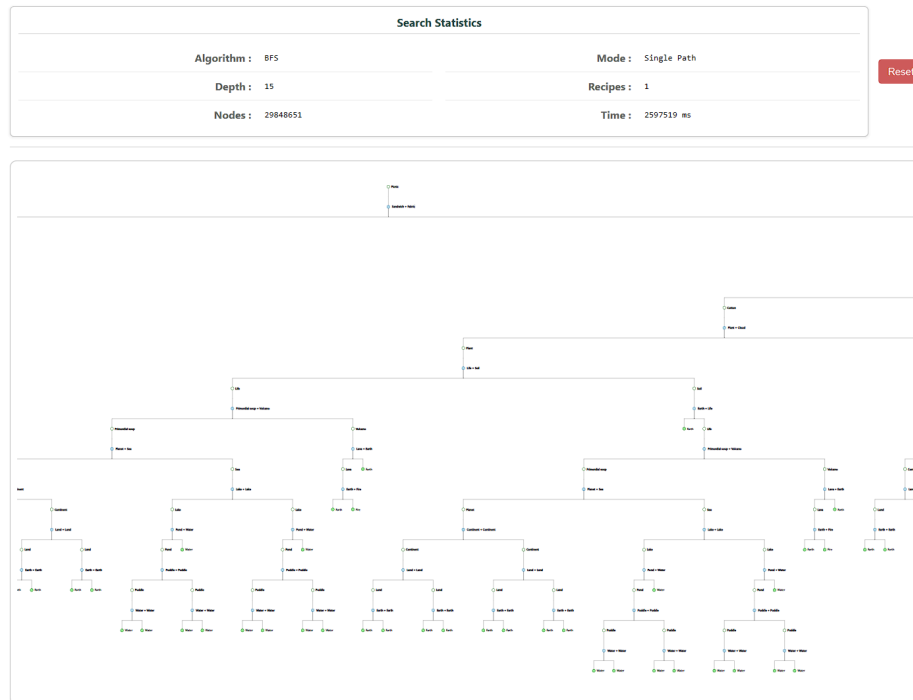
Nodes: 2479, Recipes: 2016, Time: 239 ms



IF221 Strategi Algoritma	45
--------------------------	----

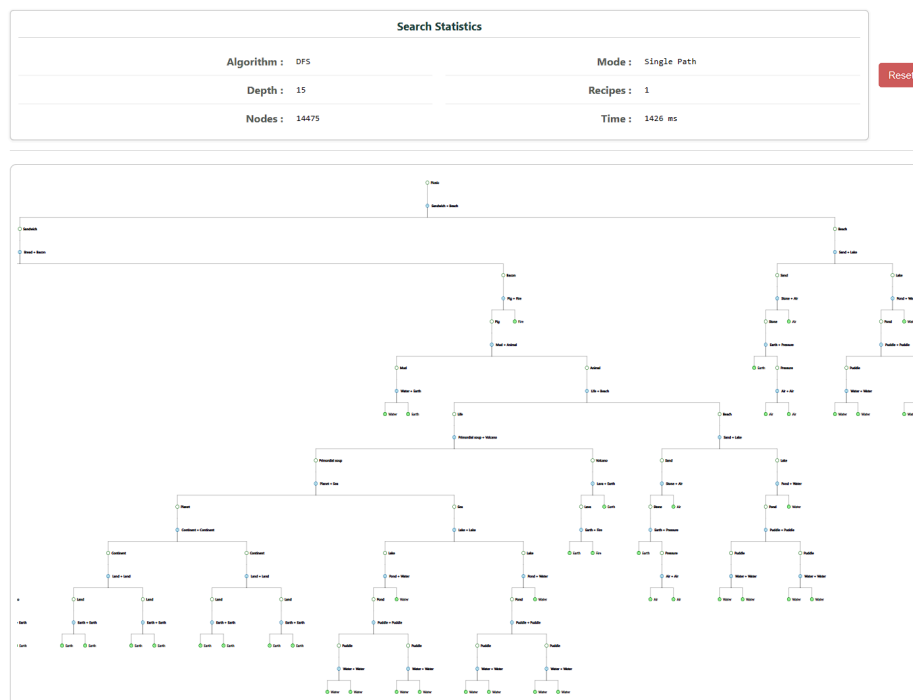
Data Uji 6: *Single Recipe* Dengan Target **Picnic**

Nodes: 29848651, Recipes: 1, Time: 2597519 ms



Gambar 4.11 Hasil BFS Data Uji 6

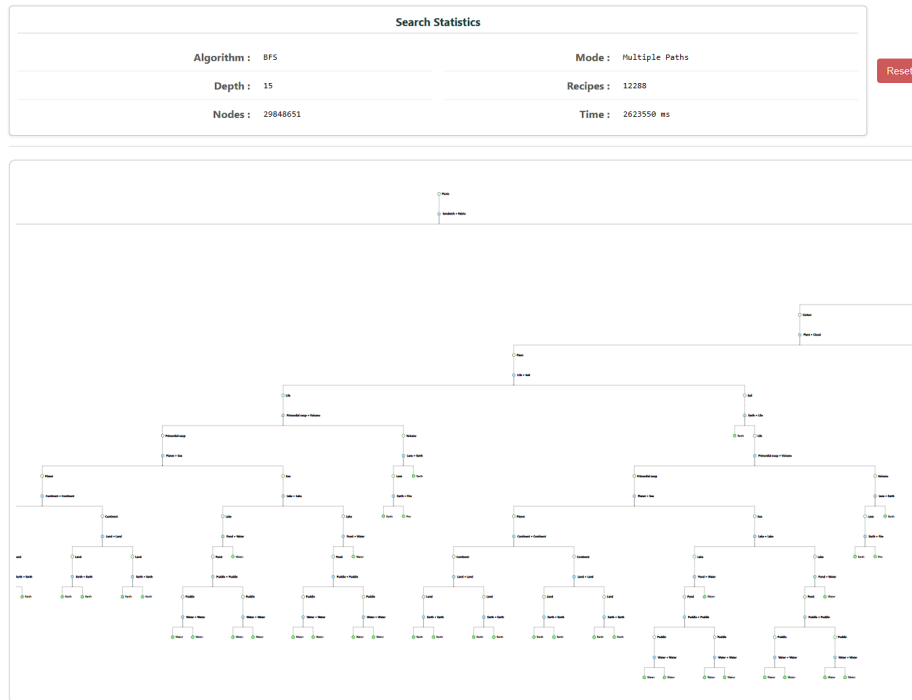
Nodes: 14475, Recipes: 1, Time: 1426 ms



Gambar 4.12 Hasil DFS Data Uji 6

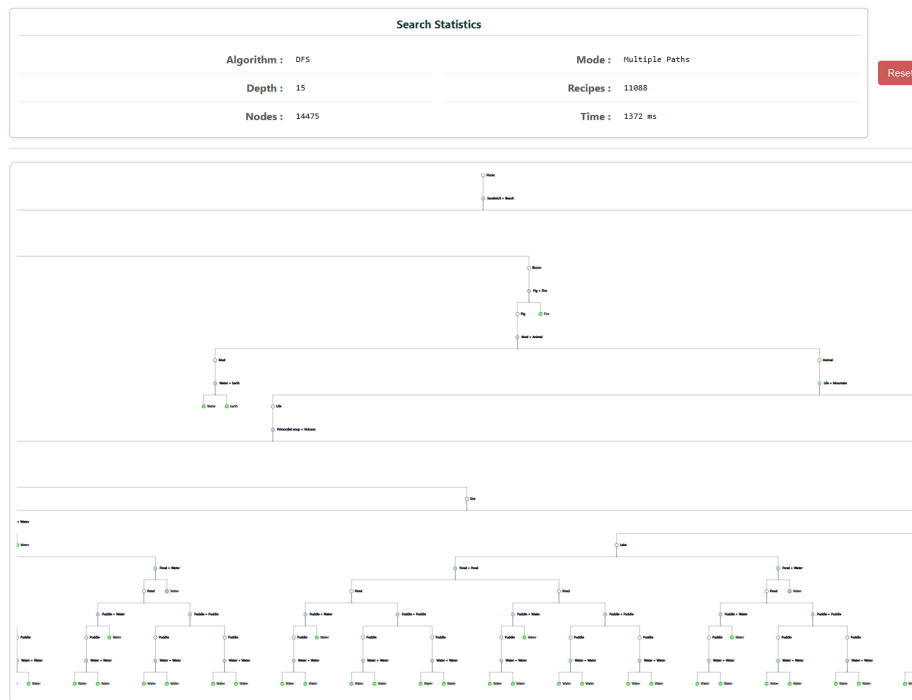
Data Uji 7: *Multiple Recipe* (11000) Dengan Target **Picnic**

Nodes: 29848651, Recipes: 12288, Time: 2623550 ms



Gambar 4.13 Hasil BFS Data Uji 7

Nodes: 14475, Recipes: 11068, Time: 1372 ms

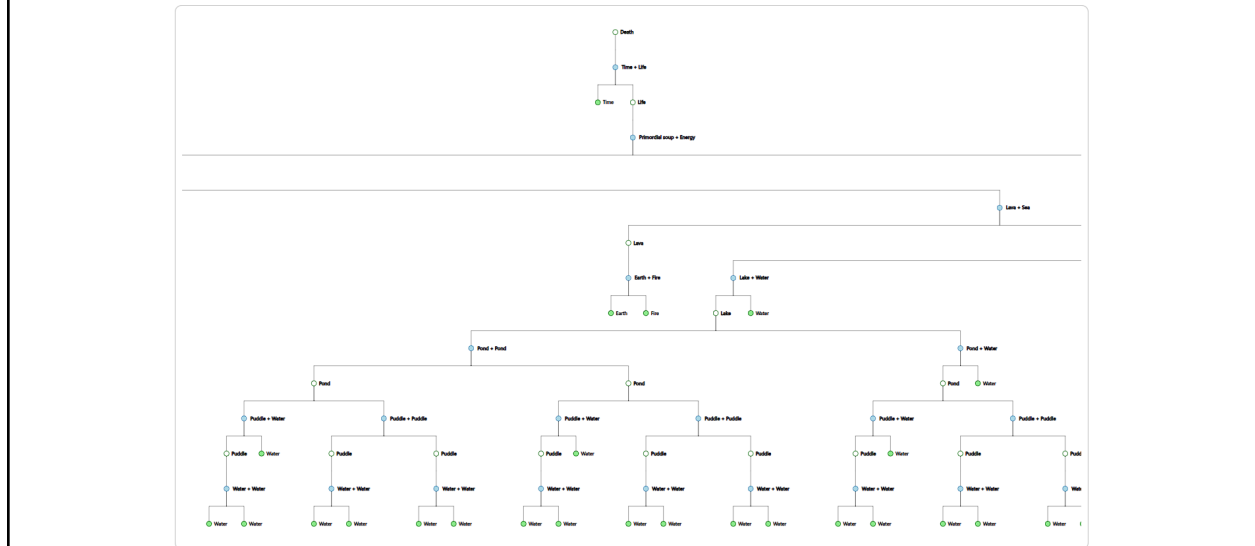


Gambar 4.14 Hasil DFS Data Uji 7

Data Uji 8: *Multiple Recipe* (69) Dengan Target **Death**

Nodes: 675, Recipes: 69, Time: 52 ms

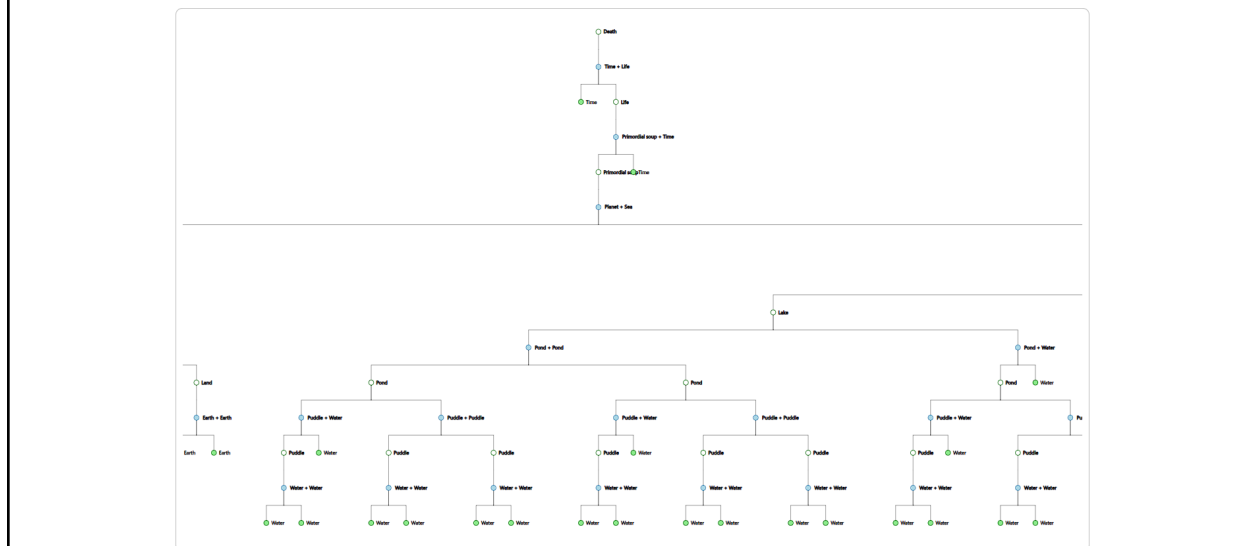
Search Statistics	
Algorithm : BFS	Mode : Multiple Paths
Depth : 7	Recipes : 69
Nodes : 675	Time : 52 ms



Gambar 4.15 Hasil BFS Data Uji 8

Nodes: 141, Recipes: 72, Time: 12 ms

Search Statistics			
Algorithm : DFS		Mode : Multiple Paths	
Depth : 7		Recipes : 72	
Nodes : 141		Time : 12 ms	



Gambar 4.16 Hasil DFS Data Uji 8

BAB V

PENUTUP

5.1. Kesimpulan

Pada Tugas Besar 2 mata kuliah IF2211 Strategi Algoritma ini, kami berhasil mengembangkan aplikasi pencarian recipe untuk elemen-elemen di permainan Little Alchemy 2. Program ini menggunakan algoritma pencarian *Breadth First Search* (BFS) dan *Depth Search First* (DFS) untuk menelusuri kombinasi elemen, sehingga dibentuk pohon resep yang merepresentasikan kombinasi tersebut. Lalu, dikembangkan *frontend* yang dapat memvisualisasikan pohon tersebut dengan baik.

Melalui pengujian yang dilakukan, sepertinya algoritma pencarian DFS memperoleh solusi jauh lebih cepat daripada BFS. Ini dikarenakan pendekatan BFS yang menelusuri simpul per level, sehingga mengunjungi banyak sekali simpul yang belum tentu membentuk solusi. Ini berbeda dengan DFS yang mengunjungi simpul secara mendalam, sehingga menyelesaikan suatu *branch* dari pohon yang dapat menjadi kandidat solusi dengan lebih cepat.

5.2. Saran

Pertama, logika untuk mendapatkan *multiple recipe* dapat ditingkatkan. Pendekatan kami untuk mencari jumlah resep sebanyak yang diinginkan user, lalu melakukan *pruning* hingga mendekati, terkadang tidak menghasilkan jumlah solusi yang pas. Pendekatan lain yang sempat dicoba takutnya terlalu menjauh dari BFS dan DFS murni, sehingga implementasinya perlu dipikirkan ulang. Ini menjadi hal yang dapat direvisikan nantinya.

Lalu, untuk pengembangan selanjutnya, sebaiknya dapat dicoba juga untuk melakukan implementasi beberapa bonus seperti implementasi seperti algoritma pencarian Bidirectional, Live Update, dan *men-deploy* aplikasi. Ini dapat membuat aplikasi web kami lebih lengkap.


5.3. Refleksi

Melalui proyek ini, kami memperoleh banyak pemahaman baru dan pemahaman mendalam mengenai implementasi algoritma pencarian BFS dan DFS. Proyek ini berhasil menunjukkan bagaimana strategi algoritma dapat diterapkan secara praktis dalam menyelesaikan permasalahan eksplorasi jalur dan kombinasi dalam permainan berbasis logika.

Kami juga belajar cara bekerja sebagai tim untuk menyelesaikan suatu proyek *full-stack*. Karena pembagian tugas *frontend* dan *backend* yang terpisah, kami dilatih cara menggabungkan hasil kerja banyak orang menjadi suatu aplikasi yang utuh.

LAMPIRAN

Link Penting

Repository	github.com/RafaAbdussalam/Tubes2_tolongpls
Video	 Tubes 2 Stima - Kelompok Tolongpls

Tabel Checkpoint

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

- "Aplikasi Berbasis Web: Pengertian, Jenis, Contoh, Keunggulan." Docif Telkom University. Diakses 12 Mei, 2025. docif.telkomuniversity.ac.id/aplikasi-berbasis-web-pengertian-jenis-contoh-keunggulan/
- "Introducing React Dev." React Blog, 16 Maret 2023. Diakses 13 Mei, 2025. react.dev/blog/2023/03/16/introducing-react-dev
- Munir, Rinaldi. "BFS-DFS (2025) Bagian 1." Program Studi Teknik Informatika, STEI ITB. Diakses 12 Mei, 2025. [informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf)
- Munir, Rinaldi. "BFS-DFS (2025) Bagian 2." Program Studi Teknik Informatika, STEI ITB. Diakses 12 Mei, 2025. [informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-\(2025\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-(2025)-Bagian2.pdf)
- "Perancangan Sistem Informasi Akademik Berbasis Web pada SMK Negeri 1 Rambah." Neliti, 2016. Diakses 12 Mei, 2025. media.neliti.com/media/publications/207585-none.pdf