

LAPORAN TUGAS BESAR 2

IF2211 STRATEGI ALGORITMA

Pemanfaatan Algoritma *BFS* dan *DFS* dalam pencarian *recipe* pada permainan *Little Alchemy 2*



Dipersiapkan oleh:

William Gerald Briandelo 13222061

Amira Izani 13523143

Anas Ghazi Al Gifari 13523159

Kelompok Go Yoon Jung

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

2025

BAB I

DESKRIPSI TUGAS



Gambar 1.1. Little Alchemy 2
(sumber: <https://www.thegamer.com>)

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

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

Pada Tugas Besar kedua Strategi Algoritma ini, mahasiswa diminta untuk menyelesaikan permainan Little Alchemy 2 ini dengan menggunakan strategi **Depth First Search** dan **Breadth First Search**. Berikut adalah komponen-komponen dari permainan ini.

1. Elemen dasar

Dalam permainan Little Alchemy 2, terdapat 4 elemen dasar yang tersedia, yaitu *water*, *fire*, *earth*, dan *air*. 4 elemen dasar tersebut akan dikombinasikan menjadi elemen turunan yang berjumlah 720 elemen.



Gambar 1.2. Elemen dasar pada Little Alchemy 2

2. Elemen turunan

Terdapat 720 elemen turunan yang terbagi menjadi beberapa *tier* tergantung pada 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*

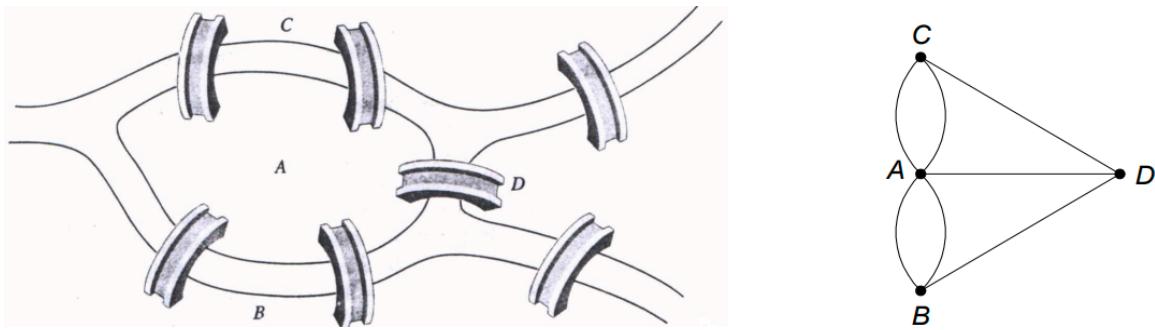
Untuk mendapatkan elemen turunan, pemain dapat melakukan kombinasi 2 elemen untuk menghasilkan elemen baru. Elemen turunan yang diperoleh dapat digunakan kembali oleh pemain untuk membentuk elemen lainnya.

BAB II

LANDASAN TEORI

2.1. Graf

Graf merupakan suatu struktur yang menggambarkan penghubungan antara himpunan elemen-elemen tidak kosong berupa titik yang disebut simpul dengan himpunan pasangan tidak terurut titik-titik tersebut yang dihubungkan oleh suatu garis yang disebut sisi. Graf digunakan untuk merepresentasikan objek-objek diskrit dan hubungan-hubungan antarobjek tersebut. Representasi visual dari graf adalah dengan menyatakan objek sebagai simpul dan hubungan antarobjek sebagai sisi.



Gambar 2.1.1. Jembatan Königsberg dan ilustrasi graf persoalannya

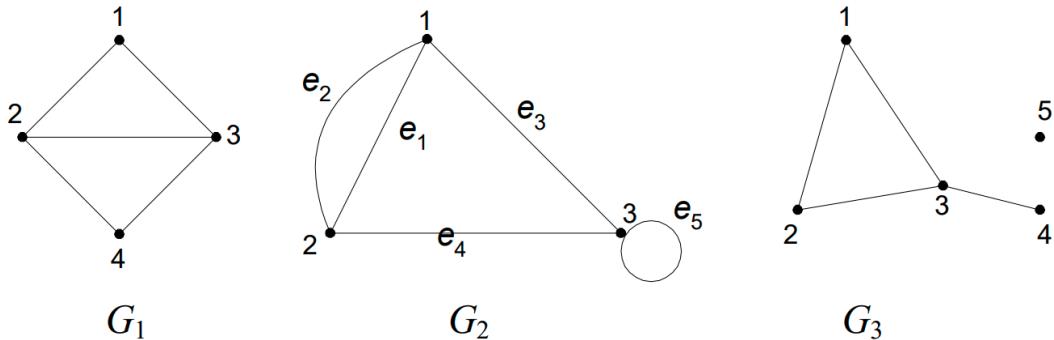
(Sumber: [20-Graf-Bagian1-2024.pdf](#))

Menurut catatan sejarah, teori graf ini pertama kali diperkenalkan oleh Swiss Leonhard Euler pada tahun 1736 untuk memecahkan permasalahan jembatan Königsberg. Euler mengilustrasikan permasalahan tersebut dalam bentuk sketsa titik dan garis yang berturut-turut merupakan representasi dari daratan dan jembatan. Ada tujuh buah jembatan yang menghubungkan daratan yang dibelah oleh sungai tersebut. Masalah pertama kali menggunakan graf ketika mencoba membuktikan kemungkinan untuk melalui ketujuh jembatan tepat satu kali. Pembuktian Euler tersebut ditulis dalam karya tulisnya yang berjudul “*Solution problematis et geometrian situ pertinensi*”.

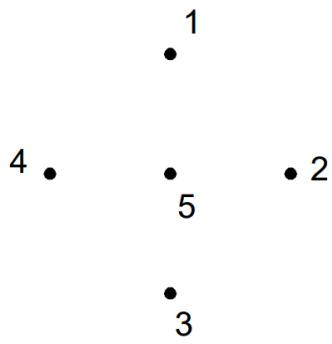
Graf G didefinisikan sebagai pasangan himpunan (V, E) , ditulis dengan notasi $G = (V, E)$, dengan V adalah himpunan tidak kosong dari simpul (*vertex*) dan E adalah himpunan sisi (*edges*) yang menghubungkan sepasang simpul. Definisi tersebut menyatakan bahwa V tidak boleh kosong, sedangkan E boleh. Simpul pada graf dapat dinyatakan dengan huruf, bilangan, atau gabungan keduanya. Sedangkan sisi-sisi yang menghubungkan simpul u dengan simpul v dinyatakan dengan pasangan (u, v) atau dinyatakan dengan lambang e_1, e_2, e_3 , dan seterusnya. Dengan kata lain, jika e adalah sisi yang menghubungkan simpul u dengan simpul v , e dapat dituliskan sebagai $e = (u, v)$.

Klasifikasi pada graf cukup luas, klasifikasi tersebut bergantung pada faktor-faktor yang membedakannya. Berdasarkan keberadaan gelang atau sisi ganda, graf digolongkan menjadi dua jenis, yaitu graf sederhana dan graf tak-sederhana. Graf tak-sederhana dapat digolongkan lagi menjadi graf ganda dan graf semu. Selanjutnya, berdasarkan orientasi arah pada sisi, graf digolongkan menjadi dua jenis, yaitu graf tak-berarah dan graf berarah.

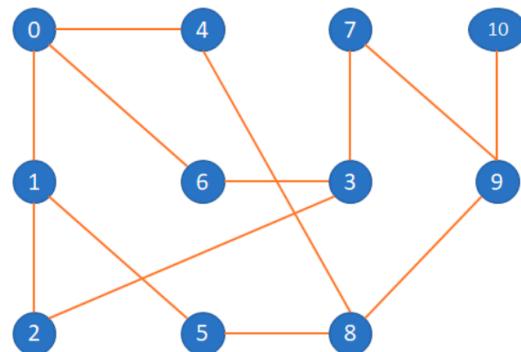
2.1.1. Terminologi Graf



Gambar 2.1.2. Contoh-contoh graf
(Sumber: [20-Graf-Bagian1-2024.pdf](#))



Gambar 2.1.3. Graf N_5
(Sumber: [20-Graf-Bagian1-2024.pdf](#))



Gambar 2.1.4. Graf sederhana
(Sumber: [20-Graf-Bagian1-2024.pdf](#))

Dalam teori graf, terdapat beberapa terminologi yang berkaitan dengan graf. Berikut akan dijelaskan beberapa terminologi tersebut satu-persatu.

1. Ketetanggaan (*Adjacent*)

Dua buah simpul pada graf G dikatakan bertetangga bila keduanya terhubung langsung dengan sebuah sisi. Dengan kata lain, u bertetangga dengan v jika (u, v) adalah sebuah sisi pada graf G . Tinjau graf G_1 pada gambar 2.1.2. Dapat dilihat bahwa simpul 1 bertetangga dengan simpul 2 dan 3. Di samping itu, dapat dilihat juga bahwa simpul 1 tidak bertetangga dengan simpul 4.

2. Bersisian (*Incidency*)

Untuk sembarang sisi $e = (u, v)$, sisi e dikatakan bersisian dengan simpul u dan simpul v . Tinjau graf G_1 pada gambar 2.1.2. Dapat dilihat bahwa sisi $(2, 3)$ bersisian dengan simpul 2 dan simpul 3, sisi $(2, 4)$ bersisian dengan simpul 2 dan simpul 4, tapi sisi $(1, 2)$ tidak bersisian dengan simpul 4.

3. Simpul Terpencil (*Isolated Vertex*)

Simpul terpencil adalah simpul yang tidak mempunyai sisi yang bersisian dengannya. Dapat dinyatakan juga bahwa simpul terpencil adalah simpul yang tidak satupun bertetangga dengan simpul-simpul lainnya. Tinjau graf G_3 pada gambar 2.1.2. Dapat dilihat bahwa simpul 5 adalah simpul terpencil.

4. Graf Kosong (*null graph* atau *empty graph*)

Graf yang himpunan sisinya merupakan himpunan kosong disebut sebagai graf kosong dan ditulis dengan notasi N_n dengan n adalah jumlah simpul. Contoh graf kosong adalah N_5 yang dapat dilihat pada gambar 2.1.3.

5. Derajat (*Degree*)

Derajat suatu simpul adalah jumlah sisi yang bersisian dengan simpul tersebut. Derajat simpul v ditulis dengan notasi $d(v)$. Amati gambar 2.1.2. Pada G_1 , dapat dilihat bahwa $d(1) = d(4) = 2$ dan $d(2) = d(3) = 3$. Pada G_3 , dapat dilihat bahwa $d(5) = 0$ yang berarti simpul 5 adalah simpul terpencil dan $d(4) = 1$. Pada G_2 , dapat dilihat bahwa $d(1) = 3$ dengan simpul 1 bersisian dengan sisi ganda dan $d(3) = 4$ dengan simpul 3 bersisian dengan sisi gelang.

6. Lintasan (*Path*)

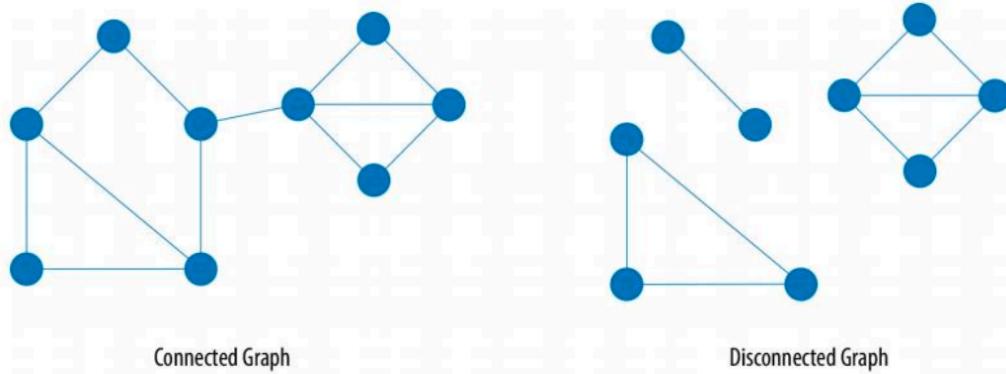
Lintasan yang panjangnya n dari simpul awal v_0 ke simpul tujuan v_n di dalam graf G adalah barisan berselang-seling simpul-simpul dan sisi-sisi yang berbentuk $v_0, e_1, v_1, e_2, v_2, \dots, v_{n-1}, e_n, v_n$ sedemikian sehingga sisi-sisi $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2), \dots, e_n = (v_{n-1}, v_n)$ merupakan anggota dari E dalam graf G . Tinjau graf pada gambar 2.1.4. Lintasan $0, 6, 3, 7, 9, 10$ adalah lintasan dari simpul 0 ke 10 yang melalui sisi $(0, 6), (6, 3), (3, 7), (7, 9), (9, 10)$ dan memiliki panjang 5.

7. Siklus (*Cycle*) atau Sirkuit (*Circuit*)

Lintasan yang berawal dan berakhir pada simpul yang sama disebut siklus atau sirkuit. Tinjau graf pada gambar 2.1.4. Lintasan $0, 4, 8, 5, 1, 0$ adalah sebuah sirkuit dengan panjang 5 karena terdapat 5 sisi dalam sirkuit tersebut.

8. Keterhubungan (*Connected*)

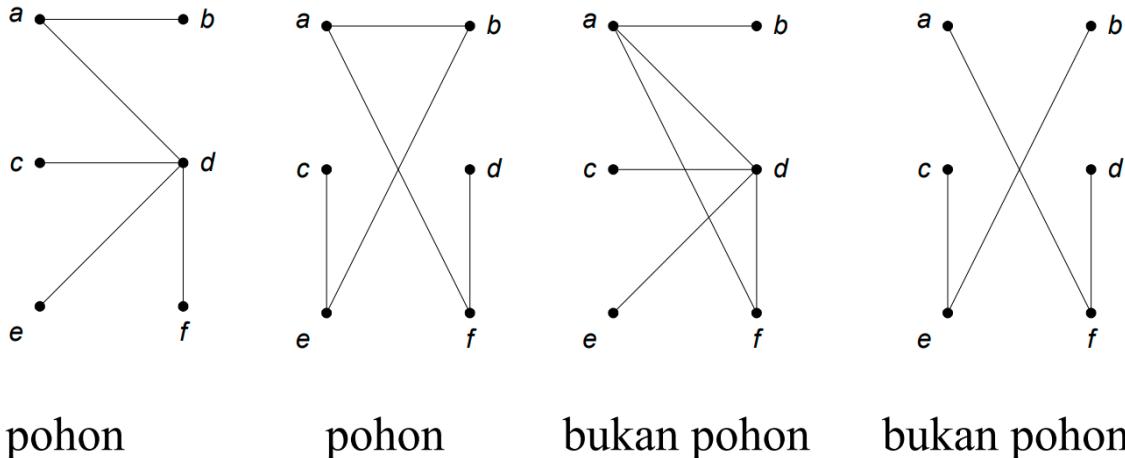
G disebut graf terhubung jika untuk setiap pasang simpul v_i dan v_j dalam himpunan V , terdapat lintasan dari v_i ke v_j . Jika tidak, G disebut graf tak-terhubung.



Gambar 2.1.5. Graf terhubung dan graf tak-terhubung
(Sumber: [20-Graf-Bagian1-2024.pdf](#))

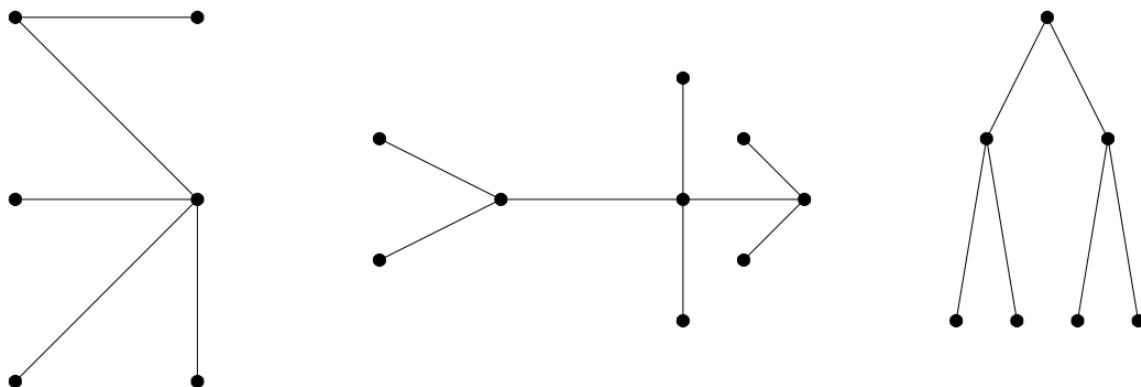
2.2. Pohon

Pohon adalah graf tak-berarah yang terhubung dan tidak mengandung sirkuit. Berikut adalah perbandingan dari graf yang merupakan pohon dan bukan pohon.



Gambar 2.2.1. Contoh pohon dan bukan pohon
(Sumber: [23-Pohon-Bag1-2024.pdf](#))

Gabungan dari beberapa pohon disebut hutan. Hutan adalah kumpulan pohon yang saling lepas atau bisa disebut juga graf tak-terhubung yang tidak memiliki sirkuit.



Gambar 2.2.2. Hutan yang terdiri dari tiga buah pohon

(Sumber: [23-Pohon-Bag1-2024.pdf](#))

Berikut merupakan sifat-sifat dari pohon. Misalkan $G = (V, E)$ adalah graf tak-berarah sederhana dan jumlah simpulnya n , maka semua pernyataan di bawah ini adalah ekivalen.

1. G adalah pohon.
2. Setiap pasang simpul di dalam G terhubung dengan lintasan tunggal.
3. G terhubung dan memiliki $m = n - 1$ buah sisi.
4. G tidak mengandung sirkuit dan memiliki $m = n - 1$ buah sisi.
5. G tidak mengandung sirkuit dan penambahan satu sisi pada graf akan membuat hanya satu sirkuit.
6. G terhubung dan semua sisinya adalah jembatan.

2.3. Penjelajahan Graf

Penjelajahan graf adalah proses mengunjungi semua simpul dalam suatu graf, yaitu struktur data yang tersusun dari simpul-simpul yang saling terhubung melalui sisi. Proses ini banyak dimanfaatkan dalam algoritma komputer untuk berbagai tujuan seperti penentuan rute, identifikasi komponen yang saling terhubung, dan analisis jaringan. Terdapat dua pendekatan umum dalam penjelajahan graf.

1. Penjelajahan secara lebar (*Breadth First Search/BFS*)

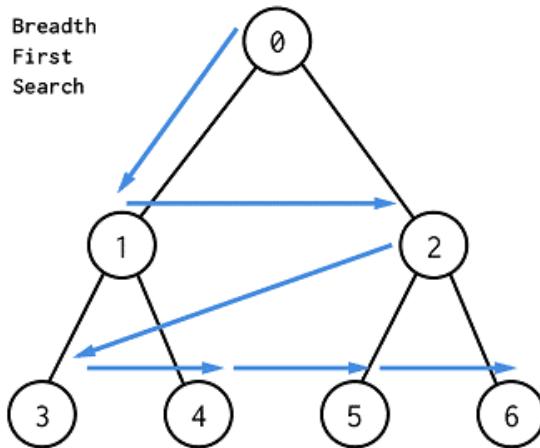
BFS bekerja dengan mengunjungi semua simpul pada tingkat kedalaman yang sama terlebih dahulu sebelum berpindah ke tingkat berikutnya. Pendekatan ini menggunakan struktur data antrian (queue) untuk melacak simpul yang akan dikunjungi berikutnya.

2. Penjelajahan secara dalam (*Depth First Search/DFS*)

DFS menjelajahi graf dengan menelusuri jalur sedalam mungkin sebelum kembali untuk menelusuri jalur lainnya. Pendekatan ini menggunakan struktur data tumpukan (stack) atau rekursi untuk melacak jalur penelusuran.

Penjelajahan graf dapat diterapkan pada graf berarah maupun graf tak-berarah. Setiap pendekatan memiliki kegunaannya sendiri tergantung pada jenis permasalahan yang dihadapi. Dalam penerapannya, penting untuk memperhatikan adanya sirkuit, komponen yang terpisah, serta simpul-simpul yang telah dikunjungi agar menghindari loop yang tak berujung.

2.4. BFS (Breadth First Search)



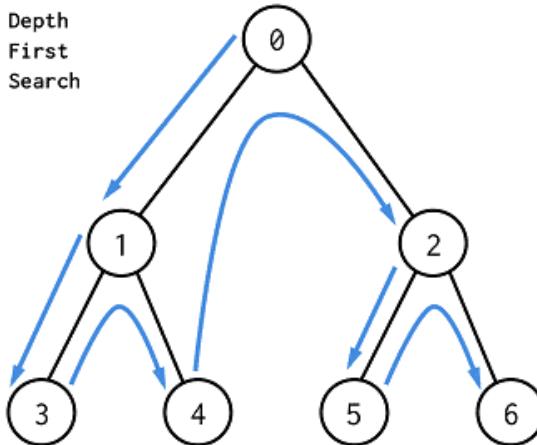
Gambar 2.4.1. Ilustrasi penjelajahan BFS

(Sumber: 2019/02/06/what-is-the-difference-between-bfs-and-dfs-algorithms)

Algoritma breadth first search (BFS) merupakan salah satu pendekatan yang digunakan untuk melakukan penelusuran pada struktur data graf untuk menemukan solusi dari suatu permasalahan. Ciri khas dari algoritma ini adalah dengan menelusuri seluruh jalur yang tersedia pada suatu tingkat kedalaman terlebih dahulu, sebelum melanjutkan ke tingkat yang lebih dalam.

Proses BFS diawali dengan memilih simpul awal, lalu mencari semua simpul yang langsung terhubung dengannya. Setelah itu, penelusuran dilanjutkan ke simpul-simpul lain yang terhubung dengan simpul-simpul sebelumnya. Dalam beberapa situasi, BFS mungkin mengunjungi kembali simpul yang sama. Namun, ketika digunakan untuk mencari jalur terpendek, simpul yang sudah dikunjungi tidak perlu diperiksa lagi. Oleh karena itu, algoritma ini memerlukan ruang penyimpanan tambahan untuk mencatat simpul-simpul yang telah ditemukan, tapi belum ditelusuri secara menyeluruh. Meskipun penggunaan memorinya lebih besar, BFS mampu memberikan solusi yang paling optimal.

2.5. DFS (Depth First Search)



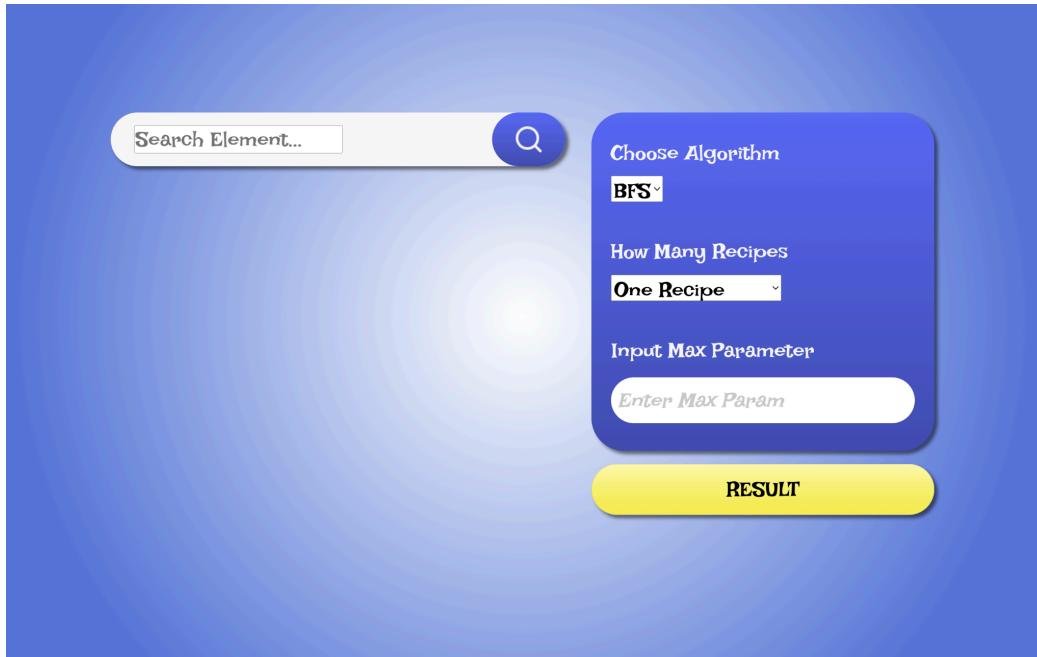
Gambar 2.5.1. Ilustrasi penjelajahan DFS

(Sumber: [2019/02/06/what-is-the-difference-between-bfs-and-dfs-algorithms](https://www.educative.io/edpresso/what-is-the-difference-between-bfs-and-dfs-algorithms))

Algoritma depth first search (DFS) merupakan salah satu pendekatan yang digunakan untuk melakukan penelusuran pada struktur data graf untuk menyelesaikan suatu permasalahan. DFS memiliki karakteristik dengan menjelajahi jalur sedalam mungkin terlebih dahulu sebelum kembali ke simpul sebelumnya untuk menjelajahi jalur lain yang belum dikunjungi.

Penelusuran DFS dimulai dari simpul awal, kemudian berpindah ke salah satu simpul tetangganya, dan terus berlanjut ke simpul berikutnya yang belum dikunjungi hingga mencapai simpul terdalam. Jika tidak ada lagi simpul yang bisa ditelusuri, algoritma akan mundur (backtrack) dan melanjutkan ke simpul lain yang belum dijelajahi. Dalam praktiknya, DFS dapat menggunakan pendekatan rekursif atau memanfaatkan struktur data tumpukan (stack) untuk mencatat jalur penelusuran. Meskipun DFS cenderung lebih hemat memori dibandingkan BFS, algoritma ini tidak selalu menjamin solusi paling optimal, terutama dalam kasus pencarian jalur terpendek.

2.6. Pembangunan Aplikasi Web



Gambar 2.6.1. Dokumentasi Recipes Finder
(Sumber: Dokumentasi Pribadi)

Aplikasi web "Recipes Finder" dirancang untuk membantu pengguna mencari recipe atau kombinasi elemen dalam permainan Little Alchemy 2, sebuah permainan yang memungkinkan pemain menggabungkan elemen-elemen dasar untuk menciptakan elemen baru. Aplikasi ini menggunakan dua metode pencarian yaitu Breadth First Search (BFS) dan Depth First Search (DFS). Pengguna dapat memilih metode pencarian yang diinginkan, menentukan elemen target, serta memilih antara pencarian satu recipe atau pencarian banyak recipe berbeda menuju elemen yang sama tersebut. Setelah itu, sistem akan memproses data kombinasi elemen hasil scraping dari wiki Little Alchemy 2, melakukan pencarian berdasarkan algoritma yang ditentukan, dan menampilkan hasil berupa visualisasi pohon recipe, jumlah node yang dikunjungi, serta waktu pencarian.

Aplikasi web ini dibangun dengan menggunakan React.js untuk antarmuka pengguna (frontend), serta Golang untuk bagian backend. Penggunaan Golang di backend memungkinkan pemrosesan algoritma pencarian dilakukan dengan cepat dan mendukung multithreading untuk optimasi pencarian multiple recipe. Visualisasi pohon recipe yang ditampilkan pada frontend menyajikan hubungan antar elemen dengan jelas. Proses scraping dinamis yang digunakan di awal pembangunan aplikasi memastikan bahwa data recipe elemen selalu akurat dan mutakhir, memungkinkan aplikasi memberikan jalur penciptaan elemen yang optimal.

2.6.1. Frontend

Frontend pada aplikasi ini berfungsi untuk menangani input pengguna dan menghasilkan visualisasi tree. Pengguna dapat memilih elemen yang tersedia dalam `recipes.json`, menentukan algoritma pencarian yang akan digunakan, yaitu BFS atau DFS, serta memilih untuk menghasilkan satu recipe atau multiple recipe. Selain itu, pengguna juga dapat menentukan jumlah recipe yang ingin dihasilkan. Setelah seluruh parameter yang telah diatur agar wajib terisi selesai diinput, pengguna dapat menekan tombol "Result". Data input tersebut akan dikirim ke backend untuk diproses, kemudian hasilnya akan ditampilkan dalam bentuk visualisasi tree berupa vertical list yang sesuai dengan pilihan pengguna. Informasi tambahan seperti jumlah node yang dihasilkan dan waktu pencarian juga akan ditampilkan. Pengguna dapat mencoba recipe lainnya dengan menekan tombol "Try More Recipe". Frontend ini dikembangkan menggunakan React.js dan TypeScript.

2.6.2. Backend

Backend pada aplikasi web dikembangkan dengan menggunakan Go. Pada backend yang dikembangkan, dilakukan inisialisasi server HTTP dan router untuk mendefinisikan endpoint yang diakses user. Backend akan mengakses data "recipes.json" yang diperoleh dari proses scraping untuk mengakses data elemen-elemen pada permainan.

Data yang diperoleh dari proses pembacaan file json tersebut, kemudian dideserialisasi menjadi struktur data Go menggunakan 'json.Unmarshal'. Selanjutnya, hasil deserialisasi tersebut akan disimpan dalam memori dan diindeks sesuai nama elemen.

Saat menerima permintaan pencarian dari endpoint, handler akan membaca parameter yang diperlukan dan memanggil metode pencarian sesuai dengan permintaan. Untuk meningkatkan performa, saat digunakan mode multi recipes, dimanfaatkan pemrosesan paralel dengan memanfaatkan sejumlah worker goroutine yang dijalankan sesuai jumlah core CPU. Setiap work pada masing-masing node parent, lalu menggabungkan hasilnya sebagai child dari node target.

Setelah algoritma menemukan satu atau beberapa jalur pembuatan elemen, hasil pencarian dikemas ulang menjadi format JSON. Respons JSON ini kemudian dikirimkan kembali kepada klien dengan header, sehingga komponen frontend dapat langsung memproses dan memvisualisasikan data dalam bentuk pohon inteker mengambil pasangan parent dari channel jobs, melakukan pencarian rekursiffraktif dengan ikon masing-masing elemen tanpa memerlukan konversi tambahan .

2.6.3. Dockerization

BAB III

ANALISIS PEMECAHAN MASALAH

3.1. Langkah-Langkah Pemecahan Masalah

Dalam program Recipes Finder yang mengimplementasikan algoritma BFS dan DFS, proses pencarian recipe untuk membentuk suatu elemen dimodelkan sebagai proses konstruksi pohon dari elemen dasar hingga elemen tujuan. Permainan Little Alchemy 2 melibatkan penciptaan elemen-elemen baru melalui penggabungan dua elemen yang telah tersedia atau telah dibuat sebelumnya. Dengan demikian, setiap elemen dapat ditelusuri asal-usul pembentukannya secara hierarkis dari elemen-elemen sebelumnya hingga akhirnya merujuk pada elemen dasar. Dalam hal ini, BFS sangat cocok untuk menemukan recipe terpendek (dalam jumlah kedalaman pohon), sementara DFS memungkinkan eksplorasi menyeluruh untuk menemukan berbagai cara pembentukan elemen. Dengan mengimplementasikan kedua algoritma tersebut, aplikasi dapat menjawab kebutuhan pencarian secara efisien, sekaligus memberikan fleksibilitas kepada pengguna untuk memilih pendekatan yang sesuai. Berikut adalah langkah-langkah pemecahan masalah terkait deskripsi program Recipes Finder.

1. Memproses masukan pengguna

Program menerima input dari pengguna berupa nama elemen target yang ingin dicari recipenya melalui antarmuka web. Selain itu, program juga menerima input jenis algoritma yang digunakan (BFS atau DFS), serta mode pencarian (one recipe atau multiple recipes). Untuk pencarian multiple recipe, pengguna juga dapat menentukan jumlah maksimal recipe yang ingin ditemukan.

2. Mengambil dan memproses data elemen

Setelah menerima input pengguna, aplikasi perlu mengambil data elemen dan recipe-nya dari hasil scraping situs wiki Little Alchemy 2. Data ini kemudian diproses menjadi struktur data pohon. Setiap elemen dianggap sebagai simpul dan anak-anak dari simpul tersebut adalah pasangan elemen yang dapat dikombinasikan untuk membentuknya. Karena satu elemen dapat memiliki lebih dari satu cara pembentukan, sebuah simpul dapat memiliki banyak cabang. Pohon ini memungkinkan sistem menelusuri jalur pembentukan dari akar (elemen target) hingga ke daun (elemen dasar).

3. Implementasi algoritma pencarian

Berdasarkan pilihan pengguna, aplikasi menerapkan algoritma pencarian yang sesuai. Jika BFS dipilih, program melakukan pencarian level demi level. BFS menjamin ditemukannya resep dengan langkah yang paling sedikit. Jika DFS dipilih, program melakukan penelusuran secara menyeluruh dan mendalam. Untuk mode multiple recipe,

pencarian dioptimalkan dengan penggunaan multithreading agar dapat menemukan berbagai jalur berbeda secara paralel dan efisien.

4. Melacak statistik pencarian

Selama proses pencarian, aplikasi mencatat statistik penting seperti jumlah simpul yang dikunjungi dan waktu eksekusi untuk memberikan transparansi dan informasi performa algoritma kepada pengguna.

5. Menampilkan hasil pencarian

Setelah pencarian selesai, aplikasi menampilkan hasil dalam bentuk visualisasi pohon yang menggambarkan langkah-langkah kombinasi elemen dari elemen dasar menuju elemen target. Cabang pohon menunjukkan pasangan elemen yang dikombinasikan dan daun pohon selalu berupa elemen dasar. Selain itu, ditampilkan juga statistik pencarian yang telah dicatat seperti jumlah simpul yang dikunjungi dan waktu eksekusi. Pembangunan hasil ini penting untuk memperlihatkan dengan jelas proses pembentukan elemen kepada pengguna.

3.2. Proses Pemetaan Masalah Menjadi Elemen-Elemen Algoritma DFS dan BFS

Proses pemetaan masalah menjadi elemen algoritma DFS dan BFS berdasarkan langkah pemecahan masalah permainan Little Alchemy 2 dilakukan sebagai berikut.

a. Node Target

Digunakan untuk melacak secara terbalik parents yang menghasilkan elemen target.

b. Node Elemen Dasar

Merupakan tujuan akhir dari pencarian dari node target.

c. Himpunan Child dan Parent

Merupakan himpunan pasangan parent dan child sepanjang jalur dari elemen target hingga elemen dasar.

Dengan memanfaatkan informasi tersebut, dapat dipetakan pemecahan masalah sebagai berikut.

3.1.1. BFS

1. Pertama akan dilakukan pengecekan apakah node tersebut memiliki parent atau tidak.
2. Jika tidak maka akan pencarian akan berhenti
3. Jika ada, maka akan ditelusuri semua parent dari elemen saat ini
4. Proses akan terus berulang hingga ditemukan pasangan parent yang sudah merupakan elemen dasar.

3.1.2. DFS

1. Pertama akan dilakukan pengecekan apakah node tersebut memiliki parent atau tidak.
2. Jika tidak maka akan pencarian akan berhenti
3. Jika ada, maka akan ditelusuri semua salah satu parent dari elemen saat ini
4. Proses akan terus berulang hingga ditemukan salah satu parent yang merupakan elemen dasar.
5. Selanjutnya, dicari pasangan dari parent-parent yang berada pada jalur tersebut dan hingga mencapai elemen dasar.
6. Lalu, dilakukan DFS ulang pada pair of parents lain hingga diperoleh jalur terpendek.

3.3. Fitur Fungsional dan Arsitektur Aplikasi Web yang Dibangun

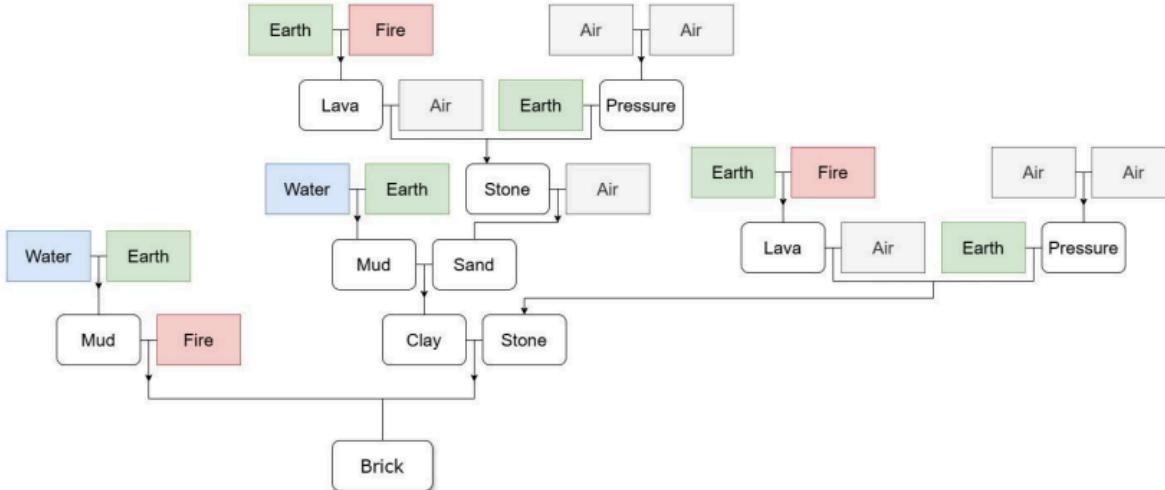
Aplikasi web dirancang untuk menyusun pohon resep kombinasi elemen dari target tertentu hingga ke elemen dasar Air, Fire, Earth, dan Water. Aplikasi ini mengandalkan arsitektur backend berbasis Golang yang efisien, dipadukan dengan API responsif yang mendukung beberapa mode pencarian dan algoritma. Pendekatan utama dalam aplikasi ini adalah pemrosesan graf non-linear dari data berbentuk JSON untuk membentuk pohon kombinasi dari elemen yang disediakan dalam struktur resep.

Struktur utama dari backend dibangun dengan mengimpor data recipes.json ke dalam struktur graf tak berarah, di mana setiap simpul merepresentasikan suatu elemen dan pasangan elemen pembentuknya. Backend menyimpan data ini dalam peta graph, yang mendukung lookup cepat untuk pencarian elemen serta relasi pembentuknya. Dua algoritma utama yang diimplementasikan adalah Depth First Search dan Breadth First Search, dengan fitur pencarian yang dikategorikan menjadi mode shortest dan multiple. Pada mode shortest, algoritma DFS dan BFS digunakan untuk mencari satu jalur terpendek dari elemen target menuju elemen dasar. Kedua metode ini menggunakan struktur Tree yang mewakili hasil pencarian dalam bentuk hierarki anak-induk. Mode multiple memanfaatkan parallel search menggunakan goroutine untuk mengeksplorasi semua kemungkinan kombinasi resep dari elemen target. Mode ini dioptimalkan dengan context cancellation dan batas waktu pencarian, serta efisiensi multithreading melalui alokasi jumlah worker berdasarkan jumlah core prosesor.

Aplikasi ini mengimplementasikan API dengan menggunakan *library* gorilla/mux, dengan endpoint utama /search mendukung tiga parameter utama: target, method, dan mode. Backend secara otomatis melakukan validasi parameter dan mengembalikan data dalam format JSON yang mencakup informasi waktu pencarian, jumlah simpul yang dikunjungi, dan hasil pohon kombinasi resep. Sistem ini juga mendukung parameter tambahan seperti limit pada mode multiple untuk mengontrol jumlah hasil pencarian. Untuk mendukung keamanan dan

kompatibilitas lintas domain, backend mengaktifkan CORS melalui pustaka handlers.CORS, yang memungkinkan integrasi frontend dari berbagai sumber tanpa hambatan akses.

3.4. Contoh Ilustrasi Kasus



Gambar 3.4.1 Contoh Visualisasi Resep

(sumber: [Spesifikasi Tubes 2](#))

3.4.1. BFS

Jika dicari resep untuk membentuk elemen Brick, maka dengan metode BFS, pertama akan diperiksa parent dari Brick, yaitu (Mud+Fire) dan (Clay+Stone). Selanjutnya, dilakukan pemeriksaan parent dari parent tersebut, hingga akhirnya ditemukan elemen dasar. Jika sudah mencapai elemen dasar maka pemeriksaan pada jalur parent tersebut akan berhenti. Namun pencarian tidak akan berhenti hingga Brick memperoleh jalur resep hingga elemen dasar.

3.4.2. DFS

Jika dicari resep untuk membentuk elemen Brick, maka dengan metode BFS, pertama akan diperiksa parent dari Brick, yaitu Mud dan akan diperiksa secara terus menerus parentnya hingga mencapai elemen dasar. Selanjutnya, akan diperiksa pair dari parent tersebut hingga tercapai jalur ke elemen dasar. Selanjutnya, akan dicari lagi jalur oleh pair of parents lain hingga semua pair of parents dikunjungi.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1. Spesifikasi Teknis Program

4.1.1. Struktur Data

```
type Recipe struct {
    Elements []string `json:"elements"`
}
```

Tipe data Recipe merepresentasikan satu resep reaksi kimia sederhana yang terdiri dari dua elemen dasar. Di dalam struktur ini terdapat satu field Elements bertipe slice of string, yang menampung nama-nama elemen yang bereaksi. Penandaan json:"elements" menjamin bahwa saat proses serialisasi atau deserialisasi JSON, susunan elemen akan dipetakan ke properti "elements" dalam berkas JSON.

Objek Recipe hanya berfokus pada pasangan unsur yang membentuk satu reaksi tunggal; validitas jumlah elemen (harus dua) baru diperiksa kemudian dalam fungsi loadRecipes, sehingga Recipe berfungsi sebagai wadah data mentah yang kemudian diproses lebih lanjut. Raw

```
type Raw struct {
    Name     string   `json:"name"`
    Recipes []Recipe `json:"recipes"`
}
```

Struktur Raw memodelkan entri resep dalam berkas JSON lengkap sebelum diolah menjadi graf. Pada Raw, Name yang menyimpan nama hasil reaksi dan Recipes yang berisi daftar alternatif resep untuk menghasilkan Name tersebut.

Sehingga, Raw digunakan sebagai medium transfer data dari file recipes.json ke dalam aplikasi, sebelum setiap entri resep dicek keunikannya dan diubah menjadi node dalam graf reaksi.

```
type Node struct {
    Name     string
    Parents [][]string
}
```

Tipe Node merupakan representasi dari satu entitas hasil reaksi dalam memori. Pada Node, Name menyimpan nama entitas, sedangkan Parents adalah slice of array yang menampung semua pasangan unsur dasar atau hasil reaksi lain yang dapat menghasilkan entitas tersebut. Setiap elemen Parents[i] berupa dua string [a, b] menandakan bahwa gabungan a dan b akan membentuk Name.

Dengan Node, aplikasi membentuk directed graph terbalik (dari produk ke reaktan), yang memudahkan traversal baik via DFS maupun BFS ketika mencari jalur reaksi menuju elemen dasar.

```
type Tree struct {
    Name      string `json:"name"`
    Children []*Tree `json:"children,omitempty"`
}
```

Struktur Tree digunakan untuk menyimpan hasil pencarian jalur reaksi yang berbentuk pohon. Name merepresentasikan node saat ini, dan Children adalah slice of pointers ke Tree lain yang mewakili langkah reaksi berikutnya. Digunakan json:"children,omitempty" untuk memastikan bahwa properti "children" hanya disertakan dalam JSON saat ada isi, sehingga daun pohon tanpa turunan tidak mencetak array kosong.

Objek Tree dihasilkan oleh bfsSearch, dfsSearch, atau multiSearch untuk menggambarkan pohon lengkap turunan reaksi dari target menuju elemen dasar, serta mudah diserialisasi kembali ke format JSON untuk respons HTTP.

```
var (
    graph      map[string]Node
    baseElements = map[string]struct{}{"Fire": {}, "Earth": {},
                                         "Air": {}, "Water": {}}
)
```

Graph adalah map yang menjadi representasi lengkap graf reaksi setelah loadRecipes dijalankan. baseElements adalah himpunan statis yang berisi elemen dasar {"Fire", "Earth", "Air", "Water"}. Digunakan untuk mengetahui kapan pencarian mencapai elemen dasar sehingga tidak diteruskan kembali.

4.1.2. Fungsi dan Prosedur

```
func loadRecipes(path string)
```

Prosedur ini berfungsi untuk memuat data resep dari file JSON yang ditentukan oleh parameter path. Data yang dimuat kemudian diurai menjadi struktur data Raw, yang berisi nama elemen dan daftar resepnya. Selanjutnya, prosedur membangun peta graf yang memetakan setiap nama elemen ke struktur Node, yang mencatat kombinasi elemen-elemen yang dapat menghasilkan elemen tersebut. Prosedur ini memastikan bahwa tidak ada duplikasi dalam kombinasi elemen yang disimpan.

```
func searchHandler(w http.ResponseWriter, r *http.Request)
```

Prosedur ini berfungsi sebagai handler untuk endpoint /search pada server HTTP. Prosdeur ini mengambil parameter target, method, dan mode dari query string permintaan HTTP. Berdasarkan parameter mode, parameter ini akan memanggil fungsi pencarian yang sesuai, seperti dfsSearch, bfsSearch, atau multiSearch. Hasil pencarian kemudian dikemas dalam format JSON dan dikirimkan sebagai respons kepada klien. Fungsi ini juga menangani validasi parameter dan mengatur header respons HTTP yang sesuai.

```
func main()
```

Prosedur ini berfungsi sebagai titik masuk utama dari aplikasi. Prosedur akan memanggil loadRecipes untuk memuat data resep dari file recipes.json. Selanjutnya, prosedur akan mengatur router HTTP menggunakan paket mux, menetapkan handler untuk endpoint /search. Prosedur ini juga mengaktifkan CORS untuk memungkinkan permintaan dari semua origin. Terakhir, server HTTP dijalankan pada port yang ditentukan oleh variabel lingkungan PORT (dengan default 8080).

```
func bfsSearch(target string) (*Tree, int64, int)
```

Fungsi bfsSearch melakukan BFS pada struktur graf berarah yang direpresentasikan oleh variabel global graf. Fungsi ini mencatat waktu mulai dan membuat node root yang merupakan target. Kemudian, sebuah queue diinisialisasi dengan objek item yang berisi node akar dan map visited agar tidak terjadi pengulangan kunjungan pada node yang sama. Setiap iterasi, node di depan queue di-dequeue, hitungan visited ditambahkan, lalu diperiksa apakah simpul tersebut termasuk dalam himpunan baseElements (kasus basis). Jika ya, fungsi langsung mengembalikan tree yang telah dibangun, durasi pencarian, dan jumlah node yang dikunjungi.

Selanjutnya, untuk setiap pasangan Parents dari simpul saat ini, fungsi memeriksa apakah masing-masing nama sudah ada di visited, jika belum, maka dua node child baru (left dan right) dibuat, dan ditambahkan ke daftar Children simpul saat ini, serta dipush ke queue dengan salinan peta visited yang diperbarui. Proses ini terus berjalan hingga queue kosong atau ditemukan elemen dasar.

```
func dfsSearch(target string) (*Tree, int64, int)
```

Fungsi ini melakukan DFS pada struktur graf berarah yang direpresentasikan oleh variabel global graf. Pencarian dimulai dengan menandai elemen target dan memanggil fungsi rekursif. Jika node yang dikunjungi merupakan elemen dasar, maka fungsi akan berhenti dan mengembalikan jalur tersebut. Jika belum menemukan elemen dasar, fungsi akan mengunjungi salah satu parent dan proses akan terus berulang hingga ditemukan elemen dasar.

```
func multiSearch(target, method string, limit int) ([]*Tree, int64, int)
```

Fungsi ini melakukan pencarian secara pararel pada setiap komponen parent dari target node. Dengan memanfaatkan multi-threading fungsi akan menyiapkan beberapa tugas dan mengerjakan tugas tersebut secara pararel sesuai jumlah core pada CPU. Pada fungsi ini akan dilakukan pemanggilan fungsi DFS atau BFS tergantung parameter yang dipilih dan fungsi akan mengembalikan beberapa resep sejumlah input yang diberikan.

4.2. Pseudocode

main.go:

Procedure LoadRecipes(path)

Kamus Lokal:

data : byte-array

```

err : error
raws : array
n : Node
a, b : string
dup : boolean

```

Algoritma:

```

data ← ReadFile(path)
if err != nil then
    FatalError(err)
endif

raws ← JSONUnmarshal(data)
graph ← empty map[string]Node

for each item in raws do
    n.Name ← item.Name
    n.Parents ← empty list
    for each r in item.Recipes do
        if length(r.Elements) = 2 then
            a ← r.Elements[0]
            b ← r.Elements[1]
            dup ← false
            for each p in n.Parents do
                if (p[0] = a and p[1] = b) or (p[0] = b and p[1] = a) then
                    dup ← true
                    break
                endif
            endfor
            if not dup then
                append [a, b] to n.Parents
            endif
        endif
    endfor
    graph[item.Name] ← n
endfor

```

Procedure SearchHandler(request, response)**Kamus Lokal:**

```

q : map[string]string
target, method, mode : string
limit : integer

```

```

resp : any
elapsed : integer
nodes : integer
lv, err : integer / error

```

Algoritma:

```

q ← ParseQuery(request.URL)
target ← q["target"]
method ← q["method"]
mode ← q["mode"]

if target = "" or method = "" or mode = "" then
    SendError(response, 400, "parameter tidak lengkap")
    return
endif

limit ← 1
if mode = "multiple" then
    (lv, err) ←ToInt(q["limit"])
    if err ≠ nil or lv ≤ 0 then
        SendError(response, 400, "batas tidak valid")
        return
    endif
    limit ← lv
endif

switch mode do
    case "shortest":
        if method = "dfs" then
            (resp, elapsed, nodes) ← DfsSearch(target)
        else
            (resp, elapsed, nodes) ← BfsSearch(target)
        endif

    case "multiple":
        (respList, elapsed, nodes) ← MultiSearch(target, method, limit)
        resp ← respList

    default:
        SendError(response, 400, "mode tidak didukung")
        return
endswitch

out ← map{
    "method": method,
}

```

```

    "mode": mode,
    "limit": limit,
    "time_us": elapsed,
    "nodes_visited": nodes,
    "recipes": resp
}
SetHeader(response, "Content-Type", "application/json")
JSONEncode(response, out)

```

Procedure Main()**Kamus Lokal:**

portStr : string
 port : string
 router : Router
 handler : HTTPHandler

Algoritma:

```

LoadRecipes("recipes.json")

portStr ← GetEnv("PORT")
if portStr = "" then
  port ← "8080"
else
  port ← portStr
endif

router ← NewRouter()
router.Handle("GET", "/search", SearchHandler)
router.ServeStatic("/", "./fe")

handler ← EnableCORS(router, origins=["*"])
Print("Server berjalan di port " + port)
ListenAndServe(": " + port, handler)

```

search.go

Function bfsSearch(target) → (pointer, integer, integer)**Kamus Lokal:**

start : timestamp
 root : pointer
 item : record { node : Tree pointer; visited : set of string }

```

queue : list of item
curr : item
count : integer
name : string
pair : [2]string
a, b : string
newVisited : set of string

```

Algoritma:

```

start ← currentTime()
root ← new Tree(Name = target)
queue ← [{ node = root, visited = { target } }]
count ← 0

while queue is not empty do
    curr ← dequeue(queue)
    count ← count + 1
    name ← curr.node.Name

    if name ∈ baseElements then
        elapsed ← timeSince(start) in microseconds
        return (root, elapsed, count)
    endif

    for each pair in graph[name].Parents do
        a ← pair[0]; b ← pair[1]
        if a ∈ curr.visited or b ∈ curr.visited then
            continue
        endif

        left ← new Tree(Name = a)
        right ← new Tree(Name = b)
        append left, right to curr.node.Children

        newVisited ← copy of curr.visited
        add a, b to newVisited

        enqueue(queue, { node = left, visited = newVisited })
        enqueue(queue, { node = right, visited = newVisited })
        break // proses satu pasangan per dequeue
    endfor
endwhile

elapsed ← timeSince(start) in microseconds
return (root, elapsed, count)

```

Function dfsSearch(target) → (pointer, integer, integer)**Kamus Lokal:**

start : timestamp
 root : pointer
 opCount : integer
 dfs : function(pointer, set of string) → boolean
 node : Tree pointer
 seen : set of string
 nextSeen : set of string
 recipe : Node record
 exists : boolean
 pair : [2]string
 left, right: Tree pointer
 success : boolean
 elapsed : integer

Algoritma:

```

start ← currentTime()
root ← new Tree(Name = target)
opCount ← 0

define dfs(node, seen):
  opCount ← opCount + 1

  if node.Name ∈ baseElements then
    node.Children ← nil
    return true
  endif

  if node.Name ∈ seen then
    return false
  endif

  nextSeen ← copy of seen
  add node.Name to nextSeen

  (recipe, exists) ← graph[node.Name]
  if not exists OR recipe.Parents is empty then
    return false
  endif

  for each pair in recipe.Parents do
    left ← new Tree(Name = pair[0])
  
```

```

right ← new Tree(Name = pair[1])
node.Children ← [left, right]

if dfs(left, nextSeen) AND dfs(right, nextSeen) then
    return true
endif

node.Children ← nil
endfor
return false
end define

success ← dfs(root, empty set)
elapsed ← timeSince(start) in microseconds
if not success then
    return (nil, elapsed, opCount)
endif
return (root, elapsed, opCount)

```

Procedure MultiSearch(target, method, limit) → (list of pointer, integer, integer)**Kamus Lokal:**

start : timestamp
 visitedCount : integer (atomic)
 node : Node
 jobs : channel of [2]string
 results : channel of Tree pointer
 ctx, cancel : context
 nWorkers : integer
 worker : concurrent function
 out : list of Tree pointer
 p : [2]string
 left, right : Tree pointer
 v1, v2 : integer
 timeout : timer

Algoritma:

```

start ← currentTime()
visitedCount ← 0
node ← graph[target]
jobs ← make channel buffered len(node.Parents)
results ← make channel buffered len(node.Parents)
(ctx, cancel) ← new cancellable context

```

```

defer cancel()

// Hitung jumlah worker sedikit di bawah CPU count
nWorkers ← computeBasedOn(runtime.NumCPU())

define worker():
    loop:
        select:
            case <- ctx.Done(): return
            case p, ok ← receive from jobs:
                if not ok then return
                if method = "dfs" then
                    (left, _, v1) ← DfsSearch(p[0])
                    (right, _, v2) ← DfsSearch(p[1])
                else
                    (left, _, v1) ← BfsSearch(p[0])
                    (right, _, v2) ← BfsSearch(p[1])
                endif

                atomic add visitedCount by (v1 + v2 + 1)
                send &Tree{Name = target, Children = [left, right]} to results
            end select
        end loop
    end define

    for i from 1 to nWorkers do spawn worker() endfor

    // Kirim pekerjaan
    spawn:
        for each p in node.Parents do send p to jobs endfor
        close jobs
    end spawn

    out ← empty list
    timeout ← after 5 seconds
    loop:
        select:
            case t ← receive from results:
                append t to out
                if length(out) ≥ limit then
                    cancel()
                    elapsed ← timeSince(start) in microseconds
                    return (out, elapsed, visitedCount)
                endif

```

```

case <- timeout:
    elapsed ← timeSince(start) in microseconds
    return (out, elapsed, visitedCount)
end select
end loop

```

4.3. Source Code

main.go:

```

package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "os"
    "strconv"

    "github.com/gorilla/handlers"
    "github.com/gorilla/mux"
)

type Recipe struct {
    Elements []string `json:"elements"`
}

type Raw struct {
    Name      string   `json:"name"`
    Recipes  []Recipe `json:"recipes"`
}

type Node struct {
    Name      string
    Parents  [][]string
}

```

```

type Tree struct {
    Name      string `json:"name"`
    Children []*Tree `json:"children,omitempty"`
}

var (
    graph      map[string]Node
    baseElements = map[string]struct{}{"Fire": {}, "Earth": {}, "Air": {},
    "Water": {}}
)

func loadRecipes(path string) {
    data, err := ioutil.ReadFile(path)
    if err != nil {
        log.Fatalf("failed to read %s: %v", path, err)
    }
    var raws []Raw
    if err := json.Unmarshal(data, &raws); err != nil {
        log.Fatalf("invalid JSON: %v", err)
    }
    graph = make(map[string]Node, len(raws))
    for _, item := range raws {
        n := Node{Name: item.Name}
        for _, r := range item.Recipes {
            if len(r.Elements) == 2 {
                a, b := r.Elements[0], r.Elements[1]
                dup := false
                for _, p := range n.Parents {
                    if (p[0] == a && p[1] == b) || (p[0] == b && p[1] ==
a) {
                        dup = true
                        break
                    }
                }
                if !dup {
                    n.Parents = append(n.Parents, [2]string{a, b})
                }
            }
        }
    }
}

```

```

    }
    graph[item.Name] = n
}
}

func searchHandler(w http.ResponseWriter, r *http.Request) {
    q := r.URL.Query()
    target := q.Get("target")
    method := q.Get("method")
    mode := q.Get("mode")
    if target == "" || method == "" || mode == "" {
        http.Error(w, "missing parameter", http.StatusBadRequest)
        return
    }

    limit := 1
    if mode == "multiple" {
        if lv, err := strconv.Atoi(q.Get("limit")); err == nil && lv > 0 {
            limit = lv
        } else {
            http.Error(w, "invalid limit", http.StatusBadRequest)
            return
        }
    }

    var (
        resp     interface{}
        elapsed int64
        nodes   int
    )

    switch mode {
    case "shortest":
        if method == "dfs" {
            t, e, v := dfsSearch(target)
            resp, elapsed, nodes = t, e, v
        } else {
    }
}

```

```

        t, e, v := bfsSearch(target)
        resp, elapsed, nodes = t, e, v
    }
    case "multiple":
        ts, e, v := multiSearch(target, method, limit)
        resp, elapsed, nodes = ts, e, v
    default:
        http.Error(w, "unsupported mode", http.StatusBadRequest)
        return
    }

    out := map[string]interface{}{
        "method":           method,
        "mode":             mode,
        "limit":            limit,
        "time_us":          elapsed,
        "nodes_visited":   nodes,
        "recipes":          resp,
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(out)
}

func main() {
    loadRecipes("recipes.json")
    port := os.Getenv("PORT")
    if port == "" {
        port = "8080"
    }
    r := mux.NewRouter()
    r.HandleFunc("/search", searchHandler).Methods("GET")
    r.PathPrefix("/").Handler(http.FileServer(http.Dir("./fe")))
    handler := handlers.CORS(handlers.AllowedOrigins([]string{"*"}))(r)
    fmt.Printf("Server started on port %s\n", port)
    log.Fatal(http.ListenAndServe(": "+port, handler))
}

```

search.go:

```
package main

import (
    "context"
    "runtime"
    "sync/atomic"
    "time"
)

func bfsSearch(target string) (*Tree, int64, int) {
    start := time.Now()
    root := &Tree{Name: target}

    type item struct {
        node      *Tree
        visited  map[string]struct{}
    }
    queue := []item{{node: root, visited: map[string]struct{}{target: true}}}
    count := 0

    for len(queue) > 0 {
        curr := queue[0]
        queue = queue[1:]
        count++
        name := curr.node.Name

        if _, ok := baseElements[name]; ok {
            return root, time.Since(start).Microseconds(), count
        }

        for _, pair := range graph[name].Parents {
            a, b := pair[0], pair[1]
            if _, seen := curr.visited[a]; seen {
                continue
            }
            if _, seen := curr.visited[b]; seen {
                continue
            }
            curr.visited[b] = struct{}{}
            queue = append(queue, item{node: &Tree{Name: b}, visited: curr.visited})
        }
    }
}
```

```

    }

    left := &Tree{Name: a}
    right := &Tree{Name: b}
    curr.node.Children = append(curr.node.Children, left, right)

    newVisited := make(map[string]struct{}, len(curr.visited)+2)
    for k := range curr.visited {
        newVisited[k] = struct{}{}
    }
    newVisited[a], newVisited[b] = struct{}{}, struct{}{}

    queue = append(queue,
        item{node: left, visited: newVisited},
        item{node: right, visited: newVisited},
    )
    break
}
}

return root, time.Since(start).Microseconds(), count
}

func dfsSearch(target string) (*Tree, int64, int) {
    start := time.Now()
    root := &Tree{Name: target}
    opCount := 0

    var dfs func(node *Tree, seen map[string]struct{}) bool
    dfs = func(node *Tree, seen map[string]struct{}) bool {
        opCount++

        if _, ok := baseElements[node.Name]; ok {
            node.Children = nil
            return true
        }

        if _, ok := seen[node.Name]; ok {
            return false
        }

        seen[node.Name] = struct{}{}
        for _, child := range node.Children {
            if !dfs(child, seen) {
                return false
            }
        }
        return true
    }

    if !dfs(root, seen) {
        return nil, 0, opCount
    }
}

```

```
    }

    nextSeen := make(map[string]struct{}, len(seen)+1)
    for key := range seen {
        nextSeen[key] = struct{}{}
    }
    nextSeen[node.Name] = struct{}{}

    recipe, exists := graph[node.Name]
    if !exists || len(recipe.Parents) == 0 {
        return false
    }

    for _, pair := range recipe.Parents {
        left := &Tree{Name: pair[0]}
        right := &Tree{Name: pair[1]}
        node.Children = []*Tree{*left, *right}

        if dfs(left, nextSeen) && dfs(right, nextSeen) {
            return true
        }
    }

    node.Children = nil
}

return false
}

success := dfs(root, make(map[string]struct{}{}))
elapsed := time.Since(start).Microseconds()

if !success {
    return nil, elapsed, opCount
}

return root, elapsed, opCount
}
```

```

func multiSearch(target, method string, limit int) ([]*Tree, int64, int)
{
    start := time.Now()
    var visited int32
    node := graph[target]
    jobs := make(chan [2]string, len(node.Parents))
    results := make(chan *Tree, len(node.Parents))
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    nWorkers := runtime.NumCPU() - (1 +
        (runtime.NumCPU() - 3) / (runtime.NumCPU() / 2))

    worker := func() {
        for {
            select {
            case <-ctx.Done():
                return
            case p, ok := <-jobs:
                if !ok {
                    return
                }
                var left, right *Tree
                var v1, v2 int
                if method == "dfs" {
                    left, _, v1 = dfsSearch(p[0])
                    right, _, v2 = dfsSearch(p[1])
                } else {
                    left, _, v1 = bfsSearch(p[0])
                    right, _, v2 = bfsSearch(p[1])
                }
                atomic.AddInt32(&visited, int32(v1+v2+1))
                results <- &Tree{Name: target, Children: []*Tree{left,
                    right}}
            }
        }
    }
}

```

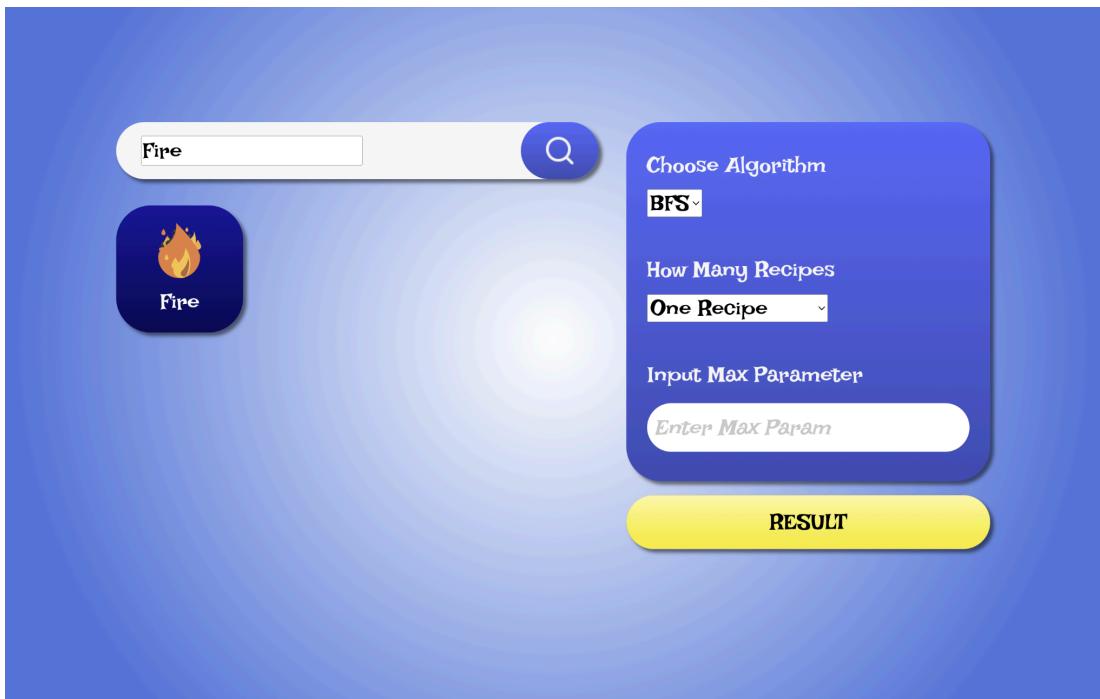
```
for i := 0; i < nWorkers; i++ {
    go worker()
}

go func() {
    for _, p := range node.Parents {
        jobs <- p
    }
    close(jobs)
}()

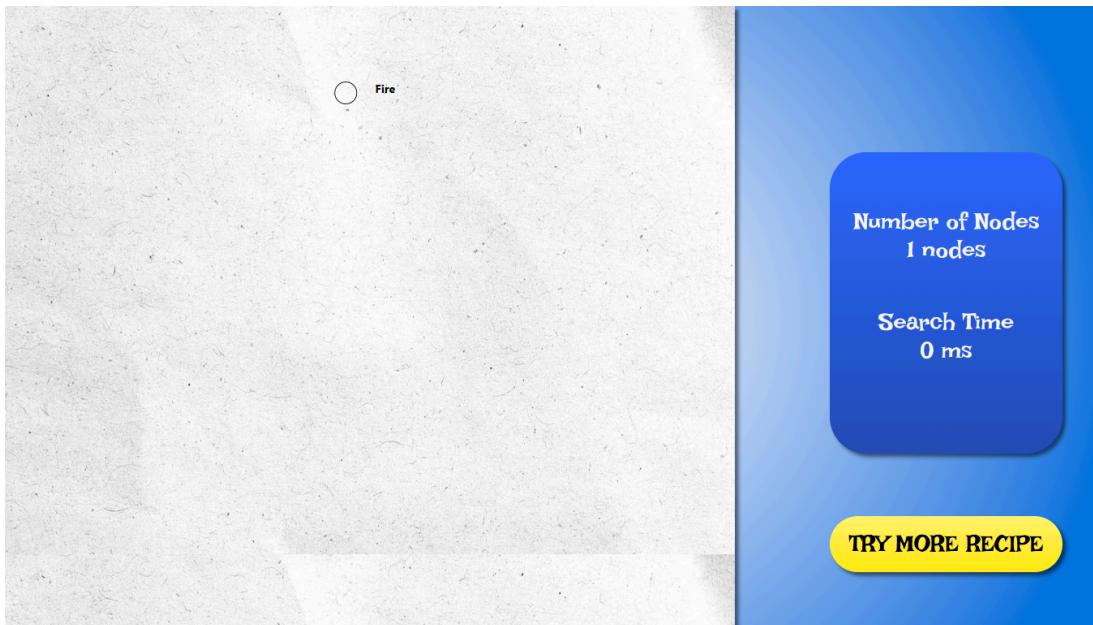
var out []*Tree
timeout := time.After(5 * time.Second)
for {
    select {
    case t := <-results:
        out = append(out, t)
        if len(out) >= limit {
            cancel()
            elapsed := time.Since(start).Microseconds()
            return out, elapsed, int(visited)
        }
    case <-timeout:
        elapsed := time.Since(start).Microseconds()
        return out, elapsed, int(visited)
    }
}
}
```

4.4. Penjelasan Tata Cara Penggunaan Program

Berikut merupakan interface yang telah dibangun



Gambar 4.4.1. Interface Input Recipe Finders
(Sumber: Dokumentasi Pribadi)



Gambar 4.4.2. Interface Output Recipe Finders
(Sumber: Dokumentasi Pribadi)

Berikut langkah-langkah penggunaan Program:

Pada interface input pengguna memasuki empat Parameter:

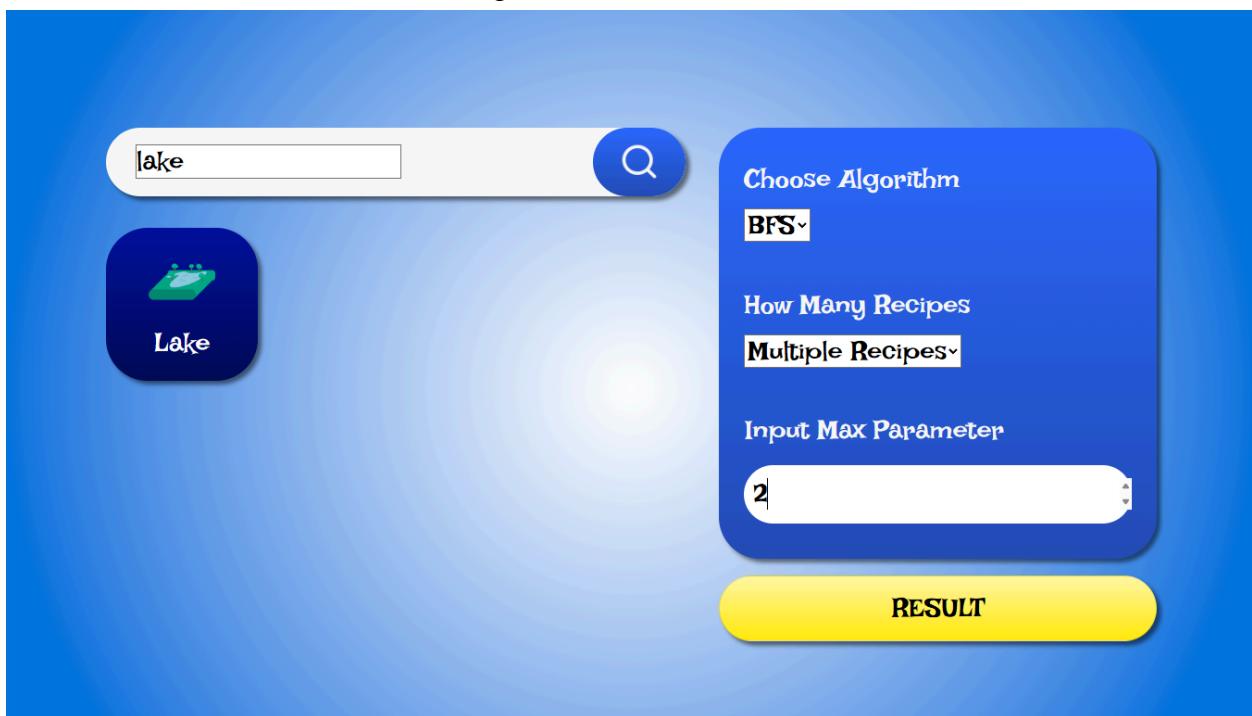
1. Pengguna memilih Algoritma antara “BFS” atau “DFS”

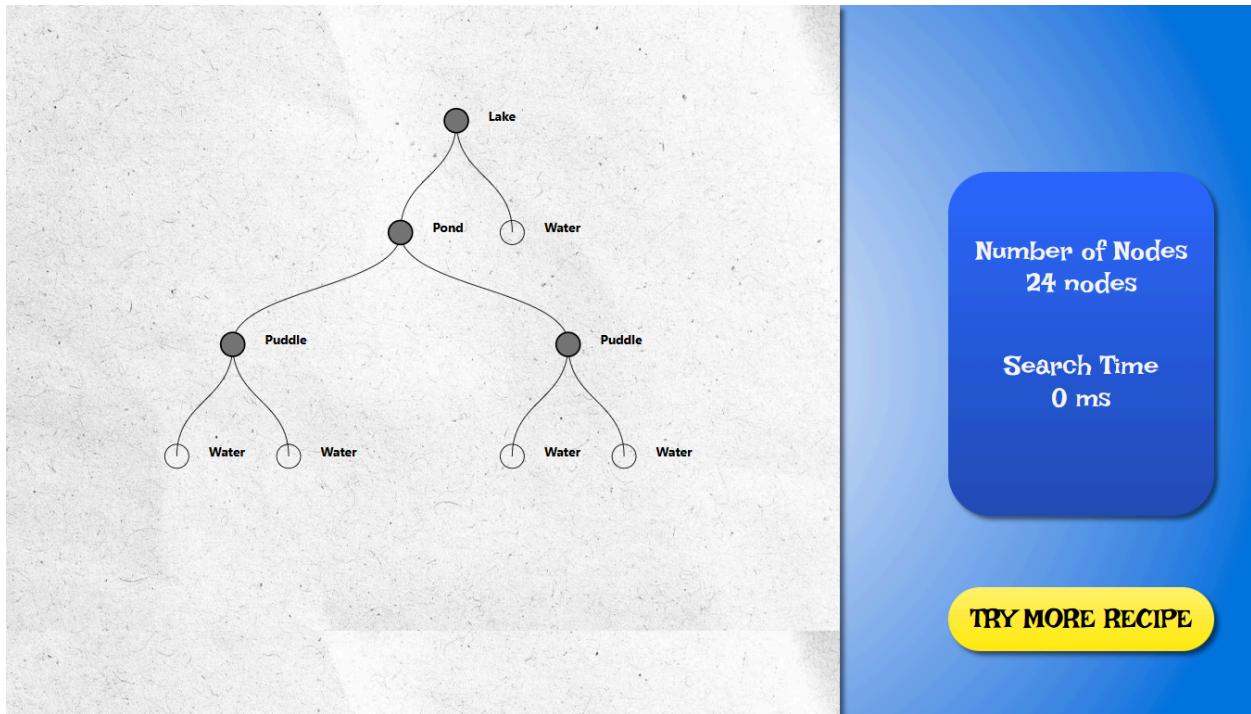
2. Pengguna memilih Jumlah recipes antara “One Recipe” atau “Multiple Recipes”
3. Apabila memilih “One recipe”, maka pengguna menginput 1 untuk Max Param dan meng-input max parameter yang diinginkan jika memilih “Multiple Recipes”
4. Pengguna menginput elemen yang diinginkan dan menekan ikon Search untuk memastikan bahwa elemen ada dalam database
5. Ketika pengguna sudah mengisi keempat parameter, pengguna bisa menekan tombol “RESULT” dan akan dipindahkan ke halaman output.

Pada interface output, pengguna bisa menekan tombol “TRY MORE RESULT” untuk mencoba recipes yang berbeda.

4.5. Hasil Pengujian

1. Percobaan 1: Lake, BFS, Multiple, 2

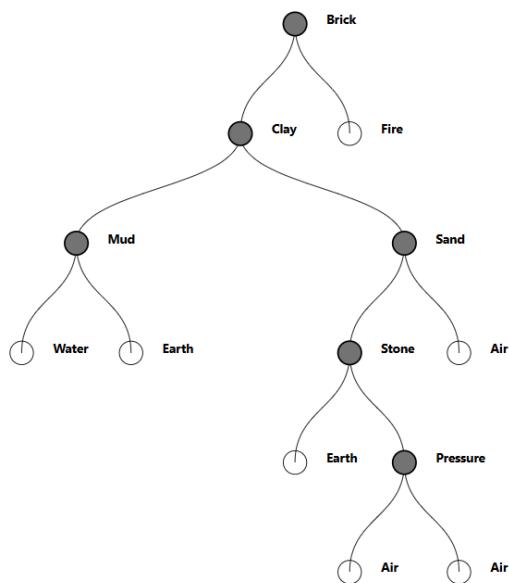
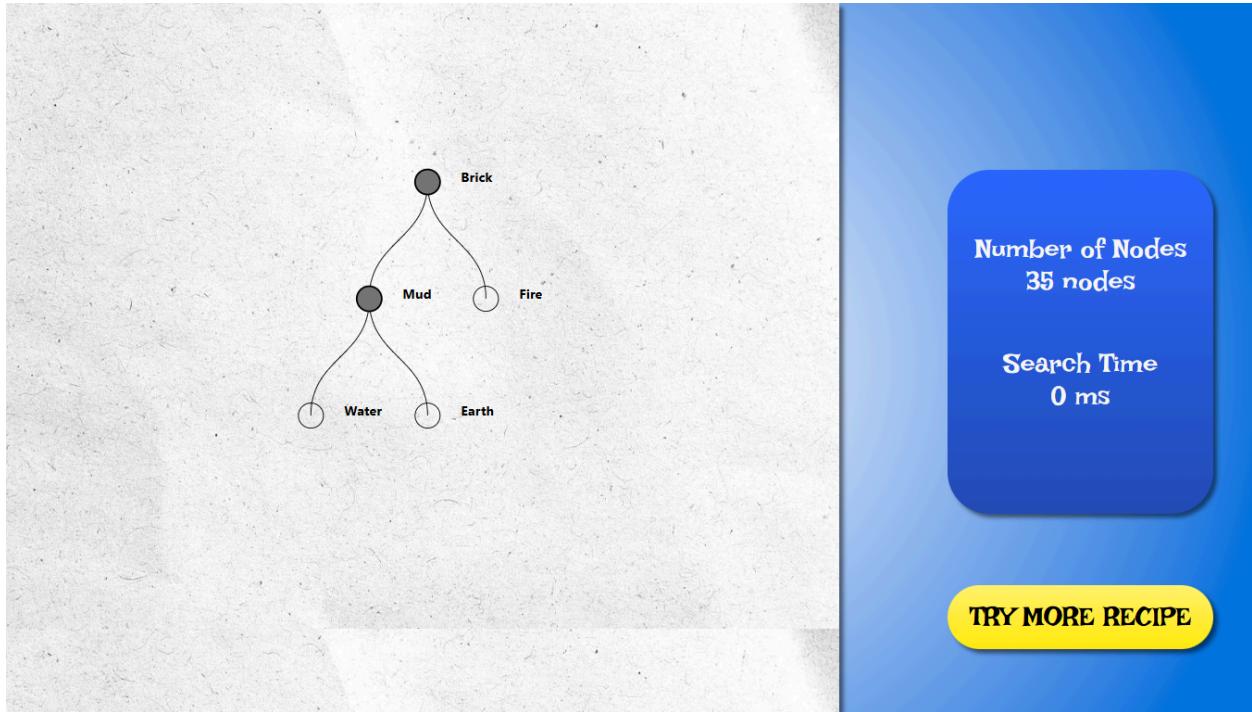


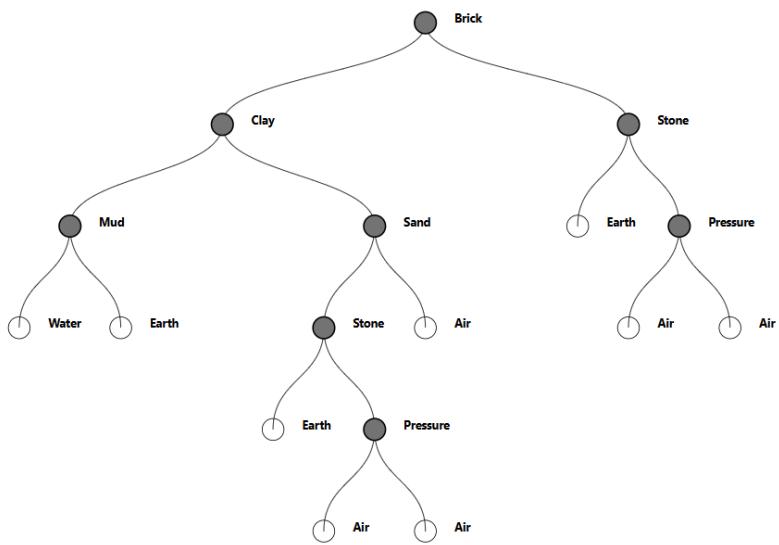


2. Percobaan 2: Brick, DFS, multiple, 3

The screenshot shows a search interface for "brick".

- Search Bar:** Contains the text "brick".
- Search Button:** A blue button with a magnifying glass icon.
- Result Preview:** A dark blue rounded rectangle containing a small image of a brick and the word "Brick".
- Algorithm Selection:** A dropdown menu labeled "Choose Algorithm" with "DFS" selected.
- Recipe Selection:** A dropdown menu labeled "How Many Recipes" with "Multiple Recipes" selected.
- Parameter Input:** A dropdown menu labeled "Input Max Parameter" with the value "3".
- Result Button:** A large yellow button labeled "RESULT".





3. Percobaan 3: Brick, BFS, one

The screenshot shows the Little Alchemy 2 interface. In the search bar at the top left, the word "brick" is typed, and a magnifying glass icon is visible. To the right of the search bar is a blue rounded rectangle containing several configuration options:

- Choose Algorithm:** BFS
- How Many Recipes:** One Recipe
- Input Max Parameter:** A slider set to 1.

A large yellow button labeled "RESULT" is positioned below these settings. On the left side of the screen, there is a small image of a brick with the word "Brick" written below it. The main central area displays a tree diagram representing the search space:

```
graph TD; Brick((Brick)) --> Mud((Mud)); Brick --> Fire((Fire)); Mud --> Water((Water)); Mud --> Earth((Earth))
```

The tree starts with "Brick" at the root. It branches into "Mud" and "Fire". "Mud" further branches into "Water" and "Earth". All elements are represented by circular icons with their names written next to them.

To the right of the search results, a blue rounded rectangle displays performance metrics:

- Number of Nodes:** 5 nodes
- Search Time:** 0 ms

A yellow button labeled "TRY MORE RECIPE" is located at the bottom right of this metrics panel.

4.6. Analisis Hasil Pengujian

Pada pengujian pencarian resep elemen dalam permainan *Little Alchemy 2*, algoritma Breadth-First Search (BFS) menunjukkan kemampuan yang efektif dalam menjelajahi seluruh jalur kombinasi resep dengan cara menyebar ke seluruh level secara merata. BFS memprioritaskan pencarian secara horizontal, yakni memperluas semua kemungkinan kombinasi pada suatu level sebelum naik ke

level berikutnya. Hal ini memungkinkan BFS untuk lebih cepat menemukan pasangan bahan yang membentuk elemen target, terutama jika pasangan tersebut terletak pada level-level awal. Dalam implementasi yang telah diperbaiki, BFS mampu membangun pohon secara lengkap hingga elemen dasar tanpa terputus, menghasilkan hasil pencarian yang lebih utuh dan sistematis.

Sebaliknya, Depth-First Search (DFS) bekerja dengan pendekatan yang berbeda, yaitu mengeksplorasi satu jalur kombinasi secara mendalam terlebih dahulu hingga mencapai elemen dasar. Baru setelah itu, DFS akan kembali (*backtrack*) dan mencoba jalur kombinasi lainnya. Strategi ini cenderung lebih lambat dalam konteks pencarian semua pasangan penyusun elemen karena DFS dapat terjebak dalam jalur yang panjang atau tidak valid sebelum menemukan solusi yang sah. Namun, DFS memiliki keuntungan dalam penggunaan memori karena hanya menyimpan jalur aktif yang sedang ditelusuri, sedangkan BFS perlu menyimpan semua node pada setiap level.

Dari segi waktu eksekusi dan kompleksitas operasional, BFS lebih unggul dalam menjelajahi dan membentuk struktur *Tree* resep secara menyeluruh dan cepat, terutama pada kasus elemen yang memiliki banyak jalur kombinasi. DFS, meskipun lebih lambat, tetap relevan untuk kasus pencarian jalur tunggal terdalam atau saat eksplorasi dibatasi kedalaman tertentu. Oleh karena itu, pemilihan metode pencarian bergantung pada kebutuhan: BFS cocok untuk membangun *recipe tree* yang lengkap, sementara DFS cocok untuk eksplorasi mendalam atau pencarian solusi unik.

BAB V

KESIMPULAN DAN SARAN

5.1. Kesimpulan

Permainan Little Alchemist 2 dapat diselesaikan dengan menggunakan algoritma BFS dan DFS yang mencari semua elemen yang menghasilkan suatu elemen lain. Penggunaan algoritma BFS dan DFS mampu menyelesaikan persoalan dengan baik, terutama algoritma BFS yang pasti menghasilkan resep dengan jalur terpendek.

Untuk permainan Little Alchemist 2, penggunaan BFS lebih baik karena menjamin diperolehnya shortest path, sedangkan untuk DFS perlu dilakukan pengulangan hingga diperoleh shortest path. Selanjutnya, untuk memperoleh multiple recipe, BFS juga mampu bekerja dengan lebih optimal dibandingkan dengan DFS. Hal ini dikarenakan pada pengerjaan BFS pada saat meninjau semua node parent dari elemen target, peninjauan tersebut juga berlaku sebagai shortest path berikutnya untuk recipe berikutnya. Namun, dengan DFS, akan ditinjau Node hingga Node elemen dasar hingga dapat memakan waktu yang sangat lama.

Selanjutnya, untuk pencarian resep, dapat juga dilakukan untuk mencari banyak resep sekaligus secara pararel dengan memanfaatkan multithreading. Hal ini bisa membuat searching lebih optimal dan bisa melihat variasi resep yang ada.

5.2. Saran

Untuk pengembangan aplikasi web ditingkatkan sehingga lebih optimal dan tidak berat. Selanjutnya, algoritma searching yang diterapkan bisa dioptimalisasi lagi sehingga waktu searching bisa lebih cepat dan hasil yang diperoleh lebih tepat. Selanjutnya, bisa dilakukan deploy pada internet, sehingga untuk akses dapat lebih mudah tanpa perlu melakukan hosting terlebih dahulu untuk setiap ingin menjalankan aplikasi web.

5.3. Refleksi

Pengerjaan Tugas Besar Strategi Algoritma sebaiknya dilakukan jauh-jauh hari sebelum deadline. Sehingga, pada saat waktu pengumpulan, pekerjaan tetap bisa berjalan lancar tanpa mengalami masalah. Dan bisa mendapat hasil yang lebih maksimal.

LAMPIRAN

Github	https://github.com/amiraIzani/Tubes2_Go-Yoon-Jung/releases/tag/v1.0
--------	-------------------------------------------------------------------------------------------------------------------------------------------------------

DAFTAR PUSTAKA

- Munir, R. (2024). *Graf (Bagian 1)*. Diakses pada 9 Mei 2025, dari
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf>
- Munir, R. (2024). *Pohon (Bagian 1)*. Diakses pada 10 Mei 2025, dari
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf>
- Munir, R. (2025). *BFS dan DFS (Bagian 1)*. Diakses pada 11 Mei 2025, dari
[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf)
- Munir, R. (2025). *BFS dan DFS (Bagian 2)*. Diakses pada 11 Mei 2025, dari
[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-\(2025\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-(2025)-Bagian2.pdf)
- FreelancingGig. (2019, Februari 6). *What is the difference between BFS and DFS algorithms?*
Diakses pada 11 Mei 2025, dari
<https://www.freelancinggig.com/blog/2019/02/06/what-is-the-difference-between-bfs-and-dfs-algorithms/>