Laporan Tugas Besar 2
IF2211 Strategi Algoritma

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

Disusun oleh:
Jovandra Otniel P S - 13523141
Muhammad Aulia Azka - 13523137
Rendi Adinata - 10123083

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025

BAB I
DESKRIPSI TUGAS

Little Alchemy 2 merupakan permainan berbasis *web* / aplikasi yang dikembangkan oleh Recloak yang dirilis pada tahun 2017, permainan ini bertujuan untuk membuat 720 elemen dari 4 elemen dasar yang tersedia yaitu *air*, *earth*, *fire*, dan *water*. Permainan ini merupakan sekuel dari permainan sebelumnya yakni Little Alchemy 1 yang dirilis tahun 2010. Mekanisme dari permainan ini adalah pemain dapat menggabungkan kedua elemen dengan melakukan drag and drop, jika kombinasi kedua elemen valid, akan memunculkan elemen baru, jika kombinasi tidak valid maka tidak akan terjadi apa-apa. Permainan ini tersedia di web browser, Android atau iOS

Pada Tugas Besar pertama Strategi Algoritma ini, akan dibuat aplikasi web dengan *front-end* dalam React dan *back-end* dalam Golang untuk menyelesaikan permainan Little Alchemy 2 ini dengan menggunakan strategi Depth First Search dan Breadth First Search.

Komponen-komponen dari permainan ini antara lain:

1. Elemen dasar Dalam permainan Little Alchemy 2, terdapat 4 elemen dasar yang tersedia yaitu water, fire, earth, dan air, 4 elemen dasar tersebut nanti akan di-combine menjadi elemen turunan yang berjumlah 720 elemen.
2. Elemen turunan Terdapat 720 elemen turunan yang dibagi menjadi beberapa tier tergantung tingkat kesulitan dan banyak langkah yang harus dilakukan. Setiap elemen turunan memiliki recipe yang terdiri atas elemen lainnya atau elemen itu sendiri.
3. Combine Mechanism Untuk mendapatkan elemen turunan pemain dapat melakukan combine antara 2 elemen untuk menghasilkan elemen baru. Elemen turunan yang telah didapatkan dapat digunakan kembali oleh pemain untuk membentuk elemen lainnya.

BAB 2
LANDASAN TEORI

Traversal Graf

Traversal graf adalah proses sistematis mengunjungi simpul dan sisi dalam struktur graf untuk melakukan pencarian rute. Graf terdiri dari simpul (*vertices*) dan sisi (*edges*), bisa berarah atau tak berarah, berbobot atau tak berbobot. Representasi umum dari graf adalah *adjacency list*, yakni upa yang menyimpan tetangga dari setiap simpul; dan *adjacency matrix*, yakni matriks biner yang menunjukkan keberadaan sisi antar-simpul.

Breadth-First Search (BFS) adalah algoritma penelusuran graf secara *level-order*, mulai dari simpul sumber, kemudian semua tetangga langsungnya, baru tetangga tingkat berikutnya. Dengan menggunakan *queue* (FIFO), setiap simpul yang dikunjungi ditandai lalu didorong ke antrian; simpul diambil dari antrian secara berurutan. BFS menjamin menemukan jalur terpendek pada graf tak berbobot. Algoritma ini memiliki kompleksitas waktu O(V+E) dan kompleksitas ruang O(V).

Depth-First Search (DFS) adalah algoritma penelusuran graf yang mengeksplorasi sedalam mungkin sebelum mundur. Biasanya diimplementasikan secara rekursif (stack implicit) atau menggunakan stack eksplisit. Mulai dari simpul sumber, DFS menelusuri jalur hingga ujung, kemudian *backtracking* untuk jalur lain. DFS efektif untuk deteksi siklus (*back-edge*) dan *topological sort* pada graf berarah *acyclic*. Algoritma ini memiliki kompleksitas waktu O(V+E) dan kompleksitas ruang O(V) (termasuk kedalaman rekursi).

Berdasarkan metode proses pencariannya BFS ideal untuk mencari solusi optimal (misalnya rute terpendek) karena memeriksa *layer by layer*. Sedangkan DFS lebih cepat dalam menemukan solusi meskipun berisiko terjebak dalam *infinite loop*. Namun, hal ini dapat diatasi dengan menambahkan syarat bahwa suatu elemen hanya dapat dibentuk dari elemen dengan *tier* di bawahnya.

Aplikasi Web

Aplikasi web ini dibangun dengan arsitektur *client–server* dengan React.js sebagai antarmuka (*frontend*) yang merender komponen dinamis, mengelola *state* dengan Context API atau Redu serta melakukan permintaan asinkron ke server menggunakan `fetch()` atau Axios tanpa perlu memuat ulang halaman. Untuk *backend*, digunakan Golang untuk menjalankan API *high-throughput* dengan memanfaatkan *concurrency* melalui *goroutine* dan *channel*, sehingga setiap permintaan pencarian resep (BFS/DFS) dapat dijalankan paralel secara efisien dan *scalable*. Komunikasi data antara kedua lapisan dilakukan melalui protokol HTTP/HTTPS dengan format JSON untuk pertukaran informasi yang mudah di-parse, sedangkan keamanan diatur melalui *middleware* CORS agar domain *frontend* dapat mengakses API Golang.

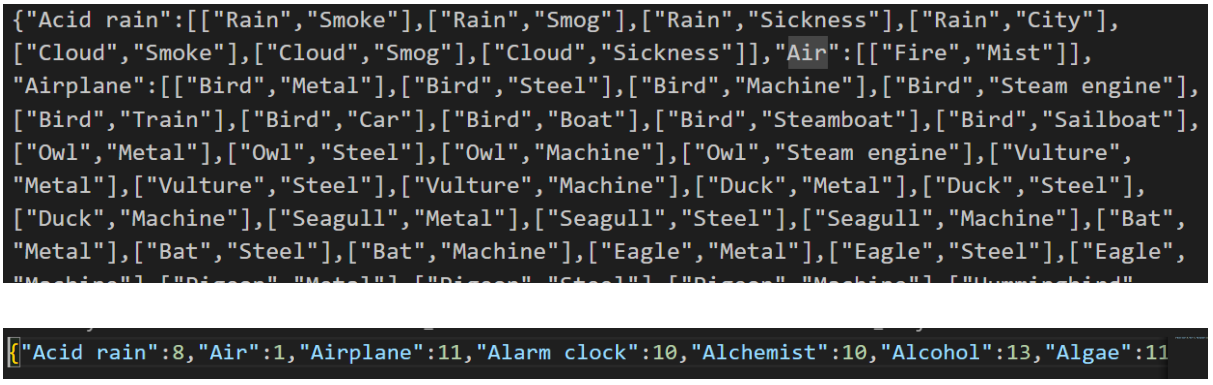
BAB 3

ANALISIS PEMECAHAN MASALAH

Untuk melakukan pencarian elemen, dilakukan terlebih dahulu *web scraping,*yakni teknik memperoleh data dari suatu web di internet.

Proses Scraping

Scraping elemen dan resep Little Alchemy 2 diimplementasikan dengan Colly di Go untuk memperoleh resep setiap elemen dan *tier* dari setiap elemen yang ditulis ke dalam berkas elements_graph.json dan elements_tier.json. Pertama-tama, setiap `<table class="list-table">` pada halaman Fandom ditandai urutannya dengan `tableIndex`, yang kemudian disimpan sebagai *tier* atau kedalaman produk di peta `tier[product] = tableIndex`. Pada setiap baris (`tbody tr`), nama produk diambil dari kolom pertama (`td:first-of-type a`), lalu setiap resep di kolom kedua (`td:nth-of-type(2) li`) diekstraksi dengan mengambil dua tag `<a>` berurutan sebagai `ingredient1` dan `ingredient2`. Pasangan bahan tersebut ditambahkan ke `graph[product] = append(graph[product], [2]string{ingredient1, ingredient2})`. Terakhir, peta `graph map[string][][2]string` untuk resep dan `tier map[string]int` untuk level masing-masing disimpan dalam berkas JSON (`elements_graph.json` dan `elements_tier.json`) sehingga *backend* dapat memuat struktur resep lengkap beserta informasi *tier* setiap elemen.



Gambar 1 Struktur data hasil *scraping* dalam elements_graph.json dan elements_tier.json

Dengan demikian, diperoleh data seperti pada gambar 1 dengan struktur berupa *dictionary* dengan *key* sebagai elemen yang dihasilkan dari beberapa *value* yang terdiri dari 2 elemen yang merupakan *ingredients*-nya. Dengan demikian, dapat dirumuskan algoritma pencarian BFS/DFS berdasarkan dataset tersebut.

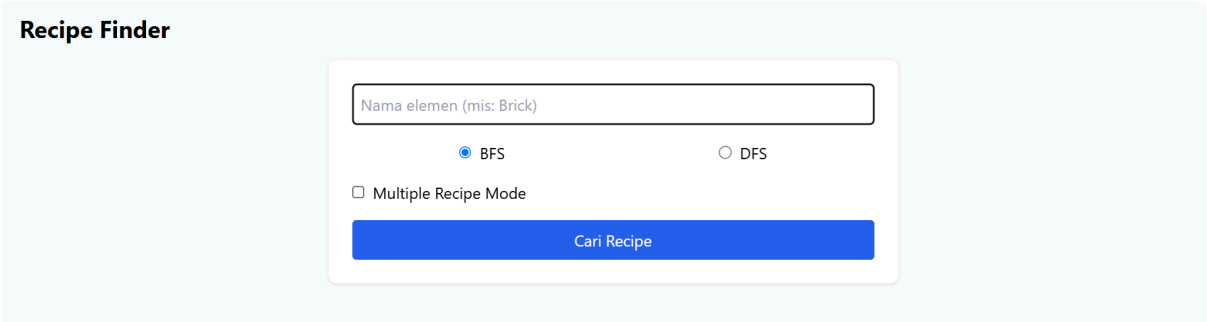
Strategi Algoritma BFS

Pada implementasi BFS, pencarian dilakukan mulai dari elemen target dan mundur menuju elemen dasar. Pertama-tama, node root `TraceNode{Product: target}` dimasukkan ke dalam *queue* dan dicatat di peta `visited`. Dalam setiap iterasi, node depan dikeluarkan, lalu semua resep `Graph[curr.Product]` diperiksa, hanya *tier* dengan bahan lebih rendah dari *tier* produk dan terbukti *buildable* (menggunakan fungsi `canBuild` dan memo `buildableMemo`) yang diproses. Untuk setiap pasangan bahan `a, b`, jika belum ada di `visited`, dibuat `TraceNode` terpisah untuk `a` dan `b`, ditambahkan ke `visited` dan di-`enqueue`. Node `curr` kemudian memperoleh informasi `From: [a, b]`, pointer `Parent` ke subtree `left` dan `right`, serta `Depth` dihitung dari kedalaman subtree, lalu `break` untuk hanya mengambil satu resep valid per produk. Proses ini berlanjut hingga *queue* kosong atau semua layer bahan dasar tercapai. Untuk mode multi-trace, `MultiBFS_Trace` memanfaatkan *goroutine* terbatas (`sem := make(chan struct{}, runtime.NumCPU())`), `sync.WaitGroup`, dan `sync.Mutex` untuk memproses setiap resep `Graph[curr.Product]` secara paralel, menghasilkan kombinasi unik dengan `hashSubtree`, mengumpulkan hingga `maxResults`, dan menyusun pohon solusi terpisah sebelum menggabungkannya kembali melalui `mergeTraceTrees`.

Strategi Algoritma DFS

Pada implementasi DFS, digunakan rekursi (LIFO) dengan *backtracking* dan *memoization* untuk membangun pohon solusi `TraceNode`. Pertama-tama, `DFS(target)` memanggil fungsi `dfsRec(prod, path)` yang mencatat jumlah simpul dikunjungi (`LastDFSVisited++`), mengecek *cache* untuk hasil sebelumnya, lalu mendeteksi siklus lewat `path[prod]`. Jika `prod` adalah elemen dasar, dibuat node baru dan di-*cache*. Untuk setiap resep `Graph[prod]` dengan bahan `a, b` yang *tier*-nya lebih rendah, dipanggil `dfsRec(a, path)`. Jika sukses tanpa siklus, `dfsRec(b, path,` jika kedua subtree valid, `TraceNode{Product: prod, From: [a,b], Parent: [left, right]}` dibuat, disimpan di *cache*, dan dikembalikan. Pada mode *multirecipe*, `MultiDFS_Trace` mengiterasi resep target, membatasi *concurrency* dengan channel `sem := make(chan struct{}, runtime.NumCPU())`, dan mem-*spawn* goroutine untuk setiap kombinasi `a×b`, lalu menggabungkan hasil yang valid hingga `maxResults` tercapai dan pada saat yang sama melindungi akses bersama dengan `sync.Mutex` dan `sync.WaitGroup`. Hasilnya adalah daftar pohon `TraceNode` unik yang siap disimpan ke JSON dan di-*render* di *frontend*.

Fitur Fungsional dan Arsitektur Aplikasi Web



Gambar 2 Tampilan antarmuka form

Aplikasi web yang dibuat memungkinkan pengguna mengeksplorasi resep pembuatan elemen Little Alchemy 2 secara interaktif dengan memasukkan nama elemen target, memilih algoritma pencarian (BFS untuk jalur terpendek, DFS untuk eksplorasi lebih dalam), mode *single* atau *multiple* (*multibfs*/*multidfs* untuk sampai n resep alternatif), dan batas `max` yang menyatakan jumlah solusi atau kedalaman. Setelah pengguna memilih "Cari Recipe", *frontend* React.js memanggil API `/api/search`, lalu hasil pencarian seperti nama elemen target, waktu eksekusi, dan jumlah simpul yang dikunjungi, ditampilkan sebagai pohon interaktif di layar menggunakan `react-d3-tree`, beserta *node* berwarna untuk membedakan elemen dasar dan turunannya.

Untuk arsitektur web, aplikasi terbagi tiga lapis yakni:

1. Lapisan data. Scraper Go dengan library Colly mengekstrak tabel resep dari situs Fandom, memfilter entri valid, dan menyimpan peta produk→pasangan bahan dalam `elements_graph.json`
2. Lapisan backend. Server HTTP Go memuat JSON ini sebagai `Graph` `map[string][][2]string` pada *startup* dan mengekspos *endpoint* `/api/search` yang menerima parameter `target`, `mode`, `algorithm`, dan `max`, lalu menjalankan algoritma BFS/DFS atau *MultiBFS*/*MultiDFS* secara paralel menggunakan goroutine, worker pool, channel, serta `sync.Mutex`/`WaitGroup` untuk keamanan konkurensi, membangun pohon `TraceNode`, dan mengembalikannya dalam JSON secara rekursif
3. Lapisan frontend. React.js mengelola *state* dan form input, memanggil API via `fetch` atau Axios, lalu memetakan JSON pohon ke komponen visualisasi untuk menampilkan struktur resep dengan interaksi *hover* dan klik yang memperluas atau memperkecil tampilan *subtree*.

Ilustrasi Kasus

Misalkan pengguna ingin membuat elemen Gold dengan maksimal tiga resep alternatif:

1. Pengguna memasukkan `target=Gold`, memilih `mode=multibfs`, lalu menetapkan `max=3`.

2. Setelah mengirim permintaan, backend memulai worker pool yang memproses node dasar secara paralel, memeriksa semua resep di graph hingga menemukan tiga jalur yang menghasilkan Gold, misalnya “Air + Metal → Gold”, “Earth + Metal → Gold”, dan “Fire + Metal → Gold”.

3. JSON yang berisi tiga pohon `TraceNode` dikembalikan, dan React me-*render* ketiganya sebagai tiga diagram pohon dalam satu tampilan dengan node tetangga dari akar menunjukkan resep ke-n.

BAB 4

IMPLEMENTASI DAN PENGUJIAN

Spesifikasi Program

Pada backend, semua resep Little Alchemy 2 disimpan dalam `RecipeGraph` tipe `map[string][][2]string` dengan setiap *key* adalah nama produk dan valuenya adalah slice pasangan dua bahan (`[2]string{"A","B"}`).

Untuk melakukan traversal graf, digunakan node `TraceNode` yang menyimpan `Product` (nama elemen), `From` (dua bahan), dan `Parent` ([2]*TraceNode) sebagai *pointer* ke *subtree*, sehingga membentuk pohon solusi lengkap yang disimpan ke dalam berkas JSON.

Pada berkas BFS.go, fungsi BFS(target string) melakukan pencarian *bottom-up* untuk membentuk elemen target dengan cara eksplorasi *backward* dari target. Algoritma ini langsung menaruh target ke dalam *queue* sebagai *root node*. Kemudian pencarian *breadth-first* pun dilakukan. Selama *queue* tidak kosong, elemen di-*dequeue* satu per satu. Untuk setiap pasangan dari Graph[curl.product], akan diperiksa apakah Tiernya lebih rendah dari Tier[curr.product] dan kedua elemen dapat dibentuk dengan fungsi canBuild(target string, tierLimit int). Jika syarat terpenuhi maka node TraceNode akan dibuat untuk a dan b (Kombinasi pasangan), ditambahkan ke *visited* dan *queue* jika belum ada. *Node* curr dikaitkan dengan a dan b melalui *Parent* dan *From*. Proses dihentikan setelah menemukan satu *recipe* valid. Proses berlanjut hingga semua *node* dalam *queue* diproses. Di bagian akhir, jika *root* tidak memiliki *parent* (tidak terbentuk dari kombinasi apapun), maka akan mengembalikan nil kecuali jika target adalah elemen dasar

Pada berkas DFS.go , traversal graf diimplementasikan secara rekursif (LIFO) dengan fungsi `dfsRec(prod, path)` yang men-*skip* siklus lewat `path[prod]`, men-*cache* hasil sukses, lalu mencoba setiap resep. Fungsi `dfsRec(a)` dan `dfsRec(b)` dipanggil dan dilakukan *backtracking* dengan unmark dan pop jika gagal, kemudian simpan *path* yang mencapai target. Untuk proses *multithreading*, dibuat fungsi `MultiBFS_Trace` dan `MultiDFS_Trace` memperluas kedua algoritma dengan *goroutine*, *worker pool*, `sync.WaitGroup`, dan `sync.Mutex` atau channel, mengumpulkan hingga `maxResults` jalur unik. MultiBFS_Trace melalui peta `nodesMap` dan ekspansi *level-order*, sedangkan MultiDFS_Trace mem-paralel-kan rekursi per resep sambil mengelola cache dan deteksi siklus. Pada startup, fungsi `LoadGraph` memuat `elements_graph.json` ke memori sebagai `Graph`, dan `SearchHandler` di `main.go` menerima request `{target, algorithm, mode, max}`, memanggil algoritma pencarian sesuai parameter, lalu mengembalikan array objek `{result, timeMs, visitedCount, tree}` sebagai JSON untuk di-render di *frontend*.

Pada berkas handler.go , program menerima permintaan POST berisi JSON `{ target, algorithm, mode, max }`, lalu mengeksekusi pencarian resep sesuai `algorithm` (`bfs` atau `dfs`) dan `mode` (`single` atau `multiple`). Pada tiap panggilan, timer dimulai, algoritma memanggil `BFS`/`DFS` (atau `MultiBFS_Trace`/`MultiDFS_Trace`), mencatat jumlah simpul yang dikunjungi, untuk BFS dari variabel global `LastBFSVisited`, untuk DFS dari `LastDFSVisited`, atau dihitung ulang dari pohon keluaran dengan `countOutputNodes`. Hasil pencarian (nama `Result`, pohon solusi `Tree`, waktu eksekusi `TimeMs`, dan `VisitedCount`) dibungkus dalam struct `SearchResponse` dan dikirim kembali sebagai array JSON. Jika target tidak ditemukan atau metode tidak didukung, handler membalas error HTTP dengan pesan yang sesuai lewat `createError`.

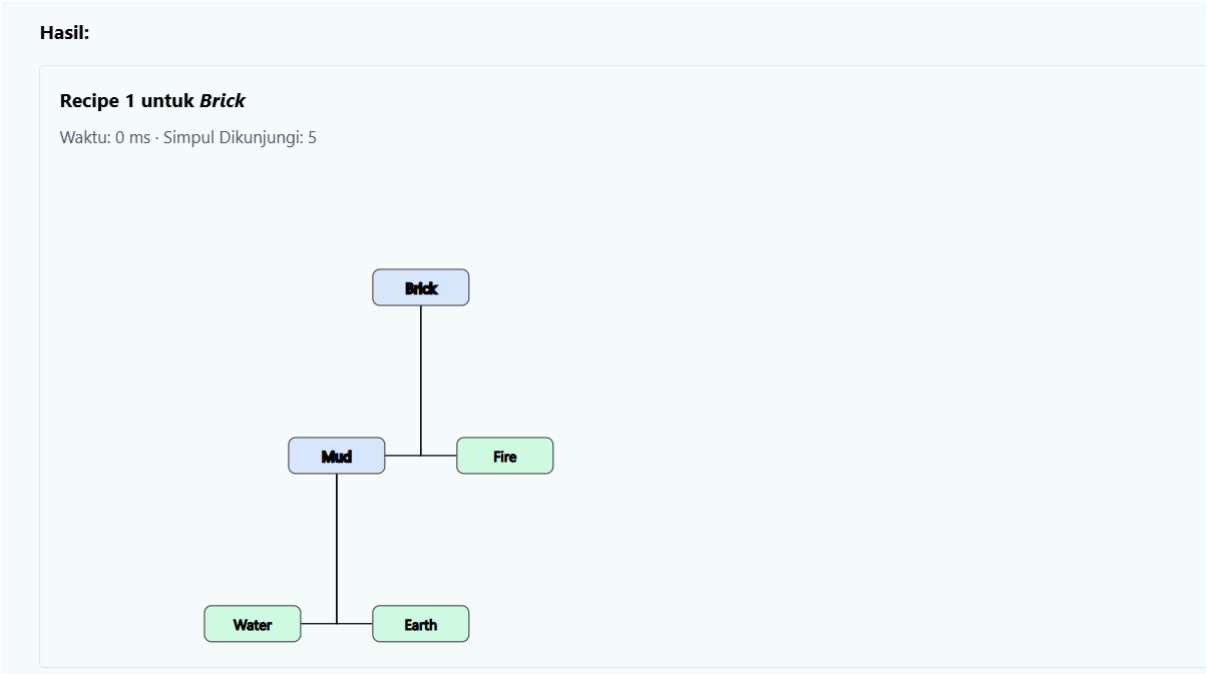
Tata Cara Penggunaan Program

1. Install Go, Node.js
2. Pada CLI, mkdir Tubes2_Goext-Chem
3. git clone https://github.com/jovan196/Tubes2_GoNext-Chem.git
4. Set Up Backend:
    1. cd src
    2. cd backend
    3. go run *.go
5. Set Up Frontend:
    1. Buka terminal baru
    2. cd src
    3. cd frontend
    4. npm install (Pastikan npm install jika baru pertama kali memulai)
    5. npm start

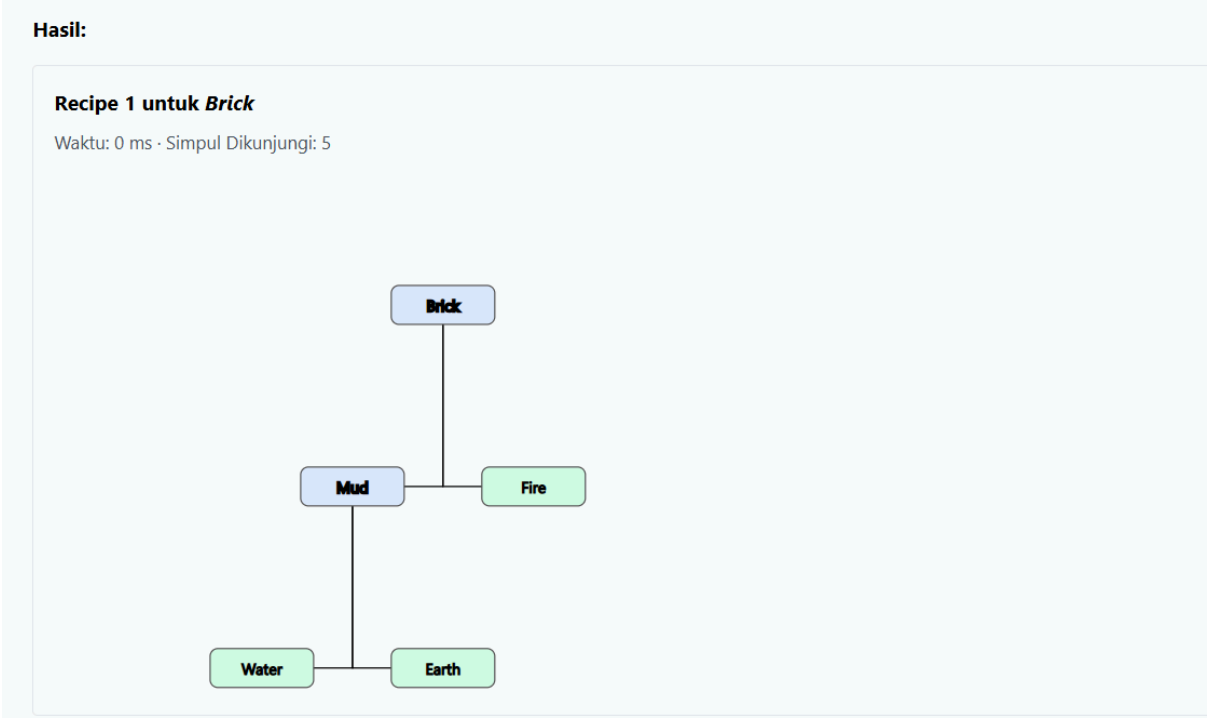
Aplikasi web akan terbuka pada peramban *default* sehingga tampilan seperti pada gambar 2 akan muncul dan pengguna dapat memasukkan parameter yang diinginkan.

Hasil Pengujian

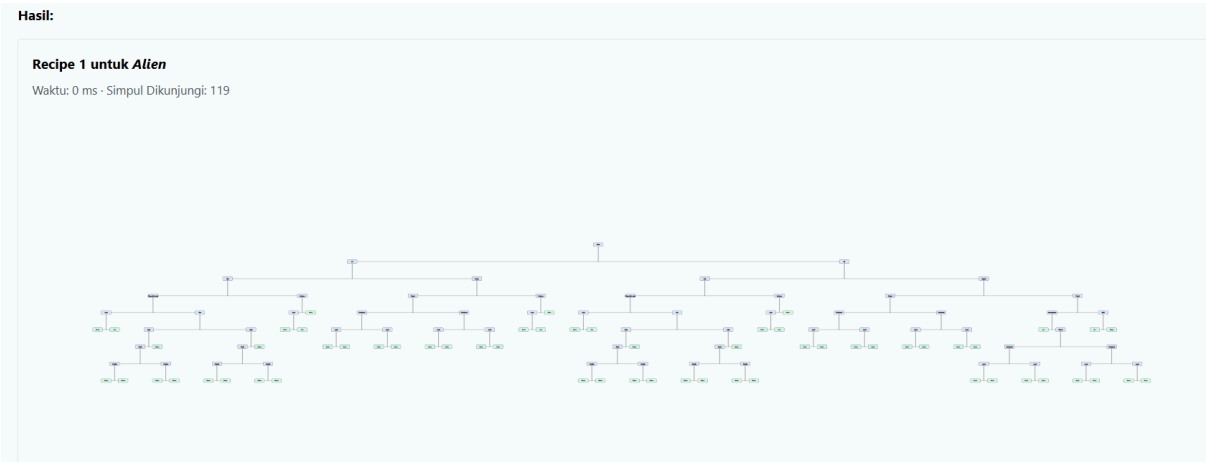
Brick: BFS - 1 Recipes



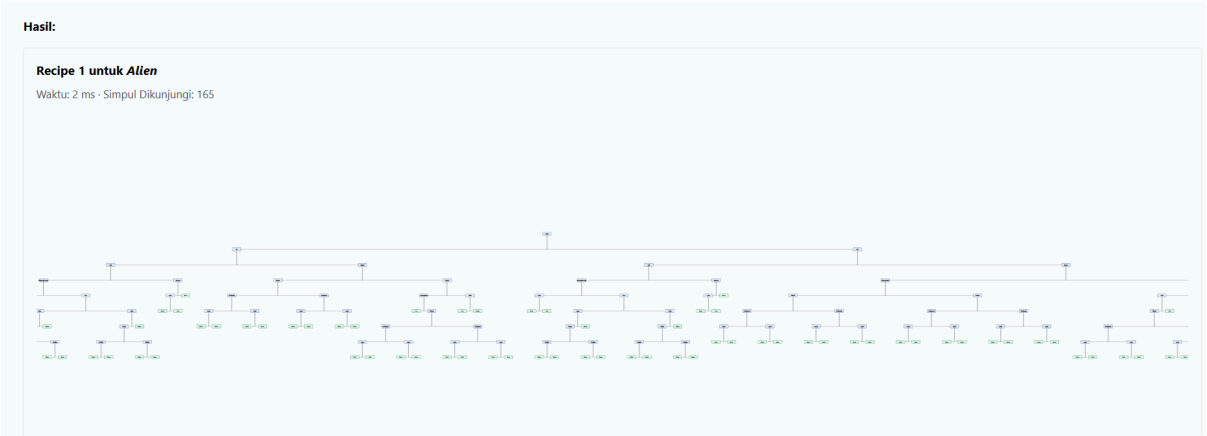
Brick: DFS - 1 Recipes



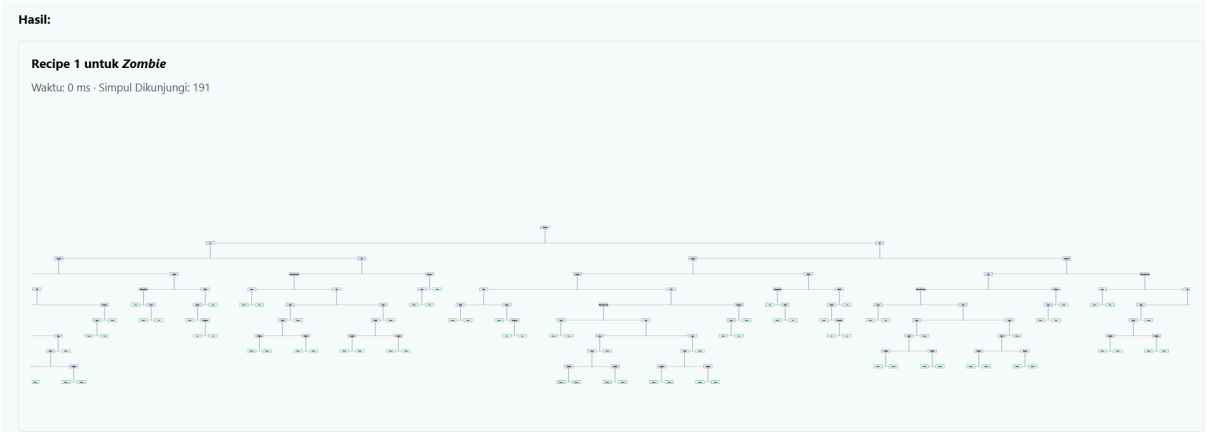
Alien : BFS- 2 Recipes



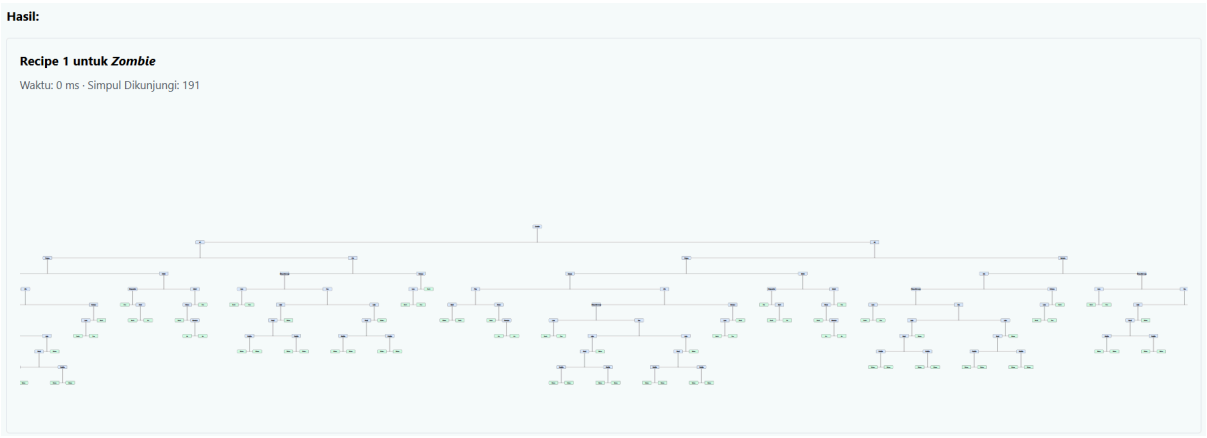
Alien : DFS - 2 Recipes



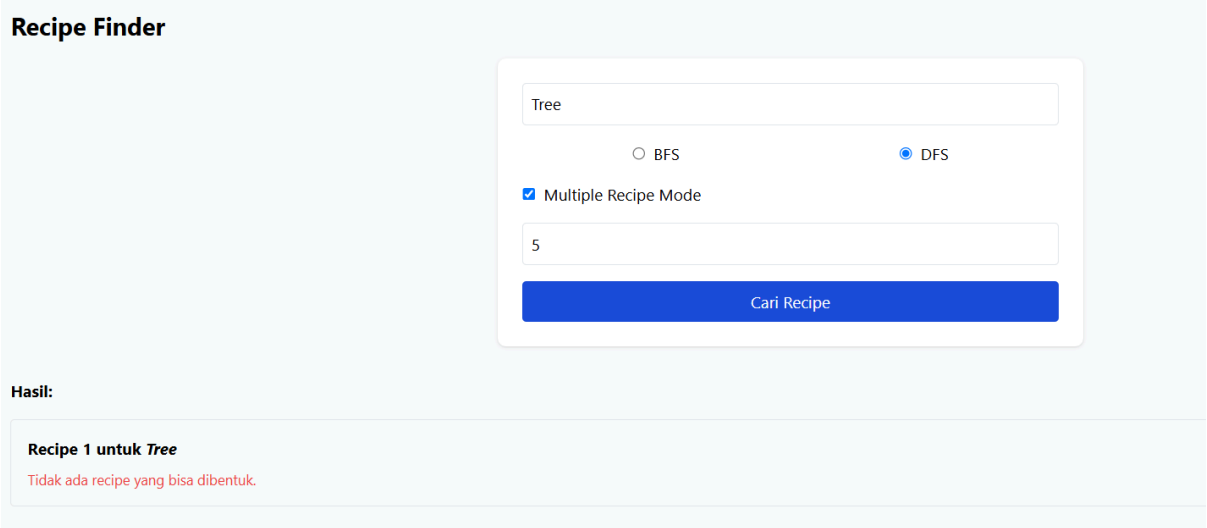
Zombie: BFS - 3 Recipes



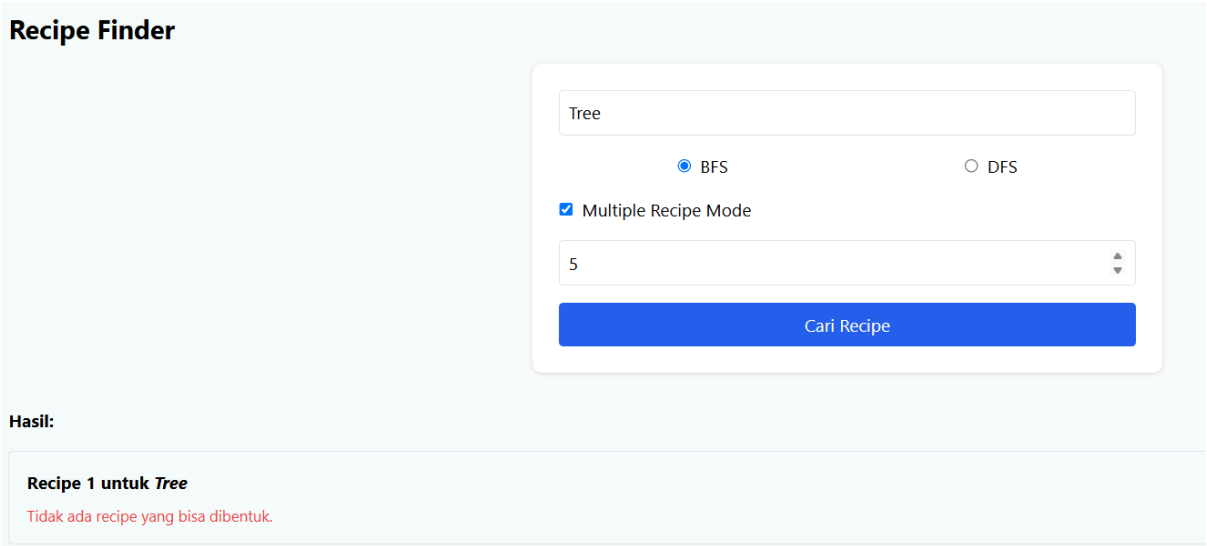
Zombie: DFS - 3 Recipes



Tree-BFS



Tree-DFS



Analisis Hasil Pengujian

Berdasarkan hasil pengujian, akan dibandingkan kecepatan pencarian dan banyaknya node yang dikunjungi antara algoritma BFS dan DFS dari ketiga elemen tersebut, yakni Brick,

Alien, Zombie, serta penjelasan mengenai mengapa ada elemen seperti Tree yang tidak ditemukan resepnya.

Pertama-tama, perhatikan bahwa optimasi pencarian menggunakan *multithreading* tidak merubah lama waktu pencarian dari 2 resep ke 3 resep secara signifikan untuk masing-masing algoritma. Ini sesuai dengan tujuan awal yakni untuk mempercepat proses pencarian secara bersamaan.

Selanjutnya, perhatikan bahwa ada beberapa elemen seperti Tree yang tidak memiliki resep apapun. Ini karena program yang dibuat akan men-*skip* elemen bahan yang *tier*-nya lebih tinggi dari elemen terbentuk. Dalam hal ini, Tree memiliki *tier* 11 dan dapat diperiksa seluruh elemen pembentuknya ada yang terdiri dari elemen dengan *tier* > 11.

Untuk waktu pencarian, menurut teori, kompleksitas waktu dari BFS dan DFS adalah O(716+3465)=O(4181), yang berarti waktu komputasi akan berbanding lurus dengan 4181 unit operasi. Hasil yang didapat sesuai dengan teori bahwa tidak terdapat perbedaan yang signifikan pada waktu pencarian elemen antara algoritma BFS dan DFS.

Selanjutnya, perhatikan bahwa node yang dikunjungi oleh BFS dari ketiga kasus tersebut pada umumnya sama atau bahkan lebih sedikit dari node yang dikunjungi DFS. Hal ini dapat disebabkan karena pencarian elemen dengan DFS dapat menggali suatu cabang hingga sangat dalam sebelum pindah ke cabang yang lainnya, sedangkan BFS akan mengeliminasi simpul-simpul yang paling dangkal terlebih dahulu.

Terakhir, perhatikan juga bahwa pohon pencarian yang dihasilkan algoritma DFS jauh lebih kompleks, sedangkan pohon yang dihasilkan BFS selalu memiliki kedalaman yang lebih rendah, bahkan minimum. Ini sejalan dengan teori bahwa selalu algoritma BFS dapat dilakukan untuk optimasi pencarian rute terpendek dalam traversal graf, dalam hal ini resep dengan elemen bahan paling sedikit.

Bab 5
KESIMPULAN

Kesimpulan

Berdasarkan implementasi algoritma pada proyek ini, dapat disimpulkan hal-hal berikut:

- Tidak terdapat perbedaan yang signifikan antara waktu pencarian algoritma BFS dan algoritma DFS
- Algoritma DFS mengunjungi lebih banyak node dari BFS untuk beberapa kasus, namun pada umumnya jumlah node yang dikunjungi sama
- Pohon pencarian BFS selalu memiliki kedalaman yang lebih rendah dibandingkan DFS
- Optimasi menggunakan *multithreading* memungkinkan pencarian banyak resep tanpa meningkatkan lama waktu pencarian secara signifikan

Saran

Pada proyek ini dapat ditambahkan fitur untuk *Live Update* agar proses pencarian dapat diamati sehingga dapat dilakukan analisis lebih mendalam. Algoritma lain seperti Bidirectional juga dapat dicoba untuk agar dapat dibandingkan efisiensinya dengan algoritma BFS dan DFS.

Refleksi

Proses integrasi *backend* (algoritma BFS/DFS) dan *frontend* (tampilan web) cukup menantang karena sinkronisasi data antara struktur pohon resep (Go) dengan visualisasi graf (React) dan optimasi performa saat menangani elemen dengan ratusan kombinasi (misal: Human, Time). Selain itu, penanganan *race condition* pada implementasi *multithreading* dan modularisasi kode untuk memisahkan logika pencarian, *scraping*, dan UI agar memudahkan kolaborasi juga menjadi tantangan dalam tubes kali ini.

REFERENSI

- informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/stima24-25.htm
- https://little-alchemy.fandom.com/wiki/Elements_(Little_Alchemy_2)
- https://go.dev/doc/
- https://github.com/PuerkitoBio/goquery

LAMPIRAN

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.		✅

pranala *repository*: jovan196/Tubes2_GoNext-Chem