LAPORAN TUGAS BESAR #2
Pemanfaatan Algoritma BFS dan DFS dalam Pencarian Recipe pada Permainan Little Alchemy 2

IF2211– Strategi Algoritma

Kelompok "*StimacinoBanataino*"

Anggota Kelompok:

Muhammad Adli Arindra	18222089
Andri Nurdianto	13523145
Hasri Fayadh Muqaffa	13523156

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

Daftar Isi

Deskripsi Tugas

Little Alchemy 2 adalah sebuah permainan kombinasi berbasis web dan aplikasi yang dikembangkan oleh *Recloak* pada tahun 2017. Dalam permainan ini, pemain memulai dengan empat elemen dasar—air (*water*), api (*fire*), tanah (*earth*), dan udara (*air*)—dan bertujuan untuk menciptakan hingga 720 elemen turunan lainnya dengan cara menggabungkan dua elemen yang tersedia. Proses kombinasi dilakukan melalui mekanisme *drag-and-drop*, yang hasilnya bergantung pada validitas kombinasi yang dilakukan. Permainan ini memberikan tantangan logika dan eksplorasi, karena elemen yang berhasil dikombinasikan dapat digunakan kembali untuk membentuk elemen yang lebih kompleks.

Tugas besar ini bertujuan untuk membangun sebuah aplikasi pencari resep elemen dalam permainan *Little Alchemy 2* dengan memanfaatkan algoritma pencarian graf yaitu *Breadth First Search* (BFS) dan *Depth First Search* (DFS). Aplikasi harus dapat menampilkan jalur pencarian dari elemen dasar hingga elemen target, baik dalam bentuk solusi tunggal maupun banyak solusi. Selain itu, aplikasi harus memberikan visualisasi berupa pohon kombinasi elemen serta informasi tambahan seperti waktu eksekusi dan jumlah simpul yang dikunjungi. Aplikasi ini dikembangkan berbasis web dengan frontend menggunakan Next.js dan backend menggunakan bahasa Go.

Laporan ini disusun untuk mendokumentasikan seluruh proses pengembangan aplikasi, mulai dari analisis permasalahan, rancangan sistem, implementasi algoritma, hingga pengujian terhadap fitur-fitur yang telah dibuat. Laporan ini juga memuat penjelasan teknis mengenai struktur data, dan arsitektur sistem. Dengan adanya laporan ini, diharapkan pembaca dapat memahami bagaimana strategi algoritma diterapkan dalam konteks permainan edukatif seperti Little Alchemy 2 serta tantangan teknis yang dihadapi dalam pengembangannya.

Landasan Teori

1. Dasar Teori

Graf adalah struktur data yang merepresentasikan sekumpulan objek (disebut simpul atau *node*) dan hubungan antar objek tersebut (disebut sisi atau *edge*). Graf dapat bersifat berarah (*directed*) maupun tak berarah (*undirected*), tergantung dari sifat relasi antar simpulnya. Dalam konteks pencarian kombinasi elemen seperti pada permainan Little Alchemy 2, setiap simpul dapat merepresentasikan sebuah elemen, sementara sisi menunjukkan kemungkinan transformasi atau kombinasi antar elemen. Untuk menjelajahi graf, dikenal beberapa metode *traversal*, yaitu proses mengunjungi simpul-simpul dalam graf, di antaranya adalah Breadth First Search (BFS) dan Depth First Search (DFS), serta Strategi Bidirectional yang digunakan untuk menemukan jalur atau struktur dari satu simpul ke simpul lainnya.

Algoritma Breadth First Search (BFS) adalah metode penelusuran graf yang menjelajah secara melebar, dimulai dari simpul awal dan mengunjungi seluruh tetangga simpul tersebut sebelum melanjutkan ke tingkat berikutnya. BFS sangat efektif untuk menemukan jalur terpendek dalam graf tak berbobot karena sifat eksplorasi level-per-level. Kelebihan BFS adalah jaminannya dalam menemukan solusi optimal dalam konteks jarak langkah terbuka. Namun, kekurangannya adalah konsumsi memori yang tinggi karena harus menyimpan seluruh simpul pada tingkat tertentu dalam struktur antrian (*queue*), yang dapat membengkak secara signifikan pada graf dengan branching factor besar.

Sebaliknya, algoritma Depth First Search (DFS) menjelajah graf dengan menyusuri jalur sedalam mungkin sebelum melakukan *backtracking*. Algoritma ini menggunakan struktur data berupa tumpukan (*stack*), baik eksplisit maupun melalui rekursi. Kelebihan DFS adalah efisiensi penggunaan memori yang relatif rendah dibandingkan BFS, serta kemampuannya dalam menjelajah seluruh kemungkinan jalur tanpa harus menyimpan semua simpul pada satu tingkat. Akan tetapi, DFS tidak menjamin pencarian solusi terpendek dan dapat mengalami eksplorasi jalur yang panjang dan tidak produktif, terutama jika solusinya berada dekat dengan akar graf atau dalam kasus graf yang sangat dalam tanpa batas eksplorasi yang jelas.

BFS dan DFS yang dijelaskan di atas hanya akan menjelajahi atau menelusuri graf dengan satu arah saja, yaitu secara maju (*forward*) atau secara mundur (*backward*). Kedua algoritma tersebut dapat dilakukan secara dua arah dengan mengimplementasikan strategi Bidirectional. Strategi Bidirectional merupakan metode penelusuran graf dari dua arah sekaligus, yaitu dari simpul awal dan dari simpul tujuan. Kedua penelusuran ini berjalan secara paralel hingga bertemu di tengah. Kelebihan dari strategi ini adalah efisiensi waktu dan ruang. Kompleksitas penelusuran yang awalnya O(b^d) dapat berkurang menjadi O(b^d/2) dengan b adalah *branching factor* dan d adalah panjang jalur terpendek.

2. Penjelasan Permainan

Permainan Little Alchemy 2 dimulai dengan empat elemen dasar: air (*water*), api (*fire*), tanah (*earth*), dan udara (*air*). Keempat elemen ini dapat dikombinasikan dua per dua untuk membentuk elemen turunan baru. Sebagai contoh, pemain dapat menggabungkan air dan tanah untuk membentuk lumpur (*mud*), lalu menggabungkan lumpur dengan api untuk menghasilkan batu bata (*brick*). Proses ini menunjukkan bagaimana dari elemen-elemen dasar saja, pemain dapat secara bertahap membentuk elemen kompleks melalui kombinasi berlapis. Elemen yang berhasil terbentuk akan tersimpan dan dapat digunakan kembali untuk membentuk elemen lainnya, memungkinkan eksplorasi yang lebih luas dalam permainan.

Satu elemen dalam permainan tidak selalu memiliki satu jalur pembuatan. Sebaliknya, banyak elemen dapat dibentuk dari berbagai kombinasi berbeda, yang disebut sebagai *recipe*. Sebagai contoh, batu (*stone*) dapat dibuat dari gabungan lava dan udara, atau dari tanah dan tekanan (*pressure*). Kondisi ini menyebabkan banyaknya kemungkinan jalur yang dapat ditempuh untuk mencapai satu elemen tertentu. Untuk mengelola kompleksitas ini, pemetaan elemen dan kombinasinya ke dalam struktur graf sangatlah membantu. Setiap elemen direpresentasikan sebagai simpul, dan setiap recipe digambarkan sebagai sisi atau jalur antara simpul. Dengan struktur ini, pencarian elemen menjadi masalah traversal graf, yang dapat diselesaikan secara efisien menggunakan algoritma seperti BFS dan DFS untuk menemukan satu atau banyak recipe menuju elemen target.

Seiring bertambahnya jumlah elemen yang telah terbentuk, kompleksitas permainan meningkat secara signifikan. Visualisasi dalam permainan Little Alchemy 2 dibuat sesederhana

mungkin untuk pemain umum—setiap elemen diwakili oleh ikon grafis dan label nama, sementara proses penggabungan cukup dilakukan dengan metode drag and drop. Namun, di balik kesederhanaan antarmukanya, terdapat struktur kombinasi yang sangat kompleks, di mana satu elemen tingkat tinggi dapat memiliki banyak jalur pembuatan dan mengandalkan hasil dari beberapa kombinasi sebelumnya. Hal ini membuat proses pencarian elemen tertentu secara manual menjadi semakin sulit, terutama jika jumlah elemen yang telah terbuka semakin banyak. Kompleksitas ini juga memunculkan kebutuhan akan pendekatan sistematis dan terstruktur seperti representasi graf untuk menavigasi ruang pencarian recipe yang luas dengan lebih efisien.

3. Penjelasan Web yang Dibangun

Frontend dikembangkan sebagai antarmuka web yang berfungsi untuk menampilkan hasil visualisasi struktur pohon (*tree*) yang diperoleh dari proses pencarian berbasis algoritma. Aplikasi web ini dibangun menggunakan *framework* **React** yang menawarkan pendekatan deklaratif dan berbasis komponen, sehingga memudahkan pengelolaan antarmuka secara modular. Untuk mendukung fitur tambahan seperti *server-side rendering* dan optimasi performa, digunakan pula **Next.js**, yang merupakan ekstensi dari React. Kombinasi React dan Next.js memungkinkan aplikasi ini untuk berjalan secara efisien dan responsif dalam berbagai perangkat dan kondisi jaringan.

Struktur pohon yang ditampilkan dalam aplikasi dibuat secara dinamis menggunakan elemen-elemen HTML dan CSS, yang dibentuk berdasarkan data berformat JSON yang diterima dari API *backend*. Proses pembentukan tree dilakukan secara rekursif, di mana setiap node dianalisis dan ditampilkan berdasarkan struktur hierarki yang ada dalam data. Posisi masing-masing node ditentukan secara relatif, agar dapat menjaga jarak dan keteraturan antar elemen dalam tampilan visual. Untuk menghubungkan antar node (*parent-child*), digunakan elemen div dengan properti border, yang kemudian dimodifikasi menggunakan transformasi CSS agar membentuk garis konektor yang sesuai, baik secara vertikal maupun diagonal.

Selain visualisasi, aplikasi ini juga menyediakan interaktivitas seperti pemilihan algoritma pencarian (*Breadth-First Search*, *Depth-First Search*, atau *Bidirectional Search*), pemilihan mode pencarian (*single* atau *multiple*), dan fitur animasi untuk

memperlihatkan proses pencarian secara bertahap. Interaktivitas ini dikendalikan menggunakan state management bawaan React, seperti useState dan useEffect, yang memungkinkan penyimpanan dan perubahan data secara real-time di dalam komponen. Seluruh interaksi pengguna secara langsung memengaruhi tampilan pohon dan memperbarui data yang ditampilkan tanpa perlu memuat ulang halaman, sehingga memberikan pengalaman pengguna yang lebih mulus.

Animasi proses pencarian pada aplikasi ini ditampilkan dengan cara memvisualisasikan urutan penemuan setiap simpul (*node*) dalam struktur pohon. Setiap simpul yang dikembalikan oleh backend telah dilengkapi dengan informasi berupa indeks penemuan, yaitu angka yang menunjukkan pada iterasi keberapa simpul tersebut ditemukan oleh algoritma pencarian. Dalam implementasinya, animasi dilakukan dengan menampilkan simpul-simpul berdasarkan urutan indeks ini secara berurutan dan bertahap, sehingga pengguna dapat melihat jalannya proses eksplorasi dari simpul awal hingga simpul target. Pendekatan ini memungkinkan pengguna memahami bagaimana algoritma menjelajahi graf secara sistematis dan memberikan visualisasi yang informatif serta edukatif mengenai cara kerja algoritma pencarian tersebut.

Untuk keperluan *deployment*, aplikasi frontend di-*hosting* menggunakan layanan Vercel, yang memiliki integrasi langsung dengan proyek Next.js dan mendukung proses *continuous deployment* melalui Git. Backend aplikasi di-*hosting* menggunakan Microsoft Azure, yang memberikan skalabilitas serta kestabilan untuk menangani permintaan API dari frontend. Kedua bagian aplikasi, baik frontend maupun backend, dikemas menggunakan Docker, sebuah platform *containerization* yang memungkinkan aplikasi berjalan dalam lingkungan yang konsisten dan terisolasi. Penggunaan Docker membantu menghindari permasalahan akibat perbedaan konfigurasi lingkungan, serta mempermudah proses pembangunan, pengujian, dan distribusi aplikasi dalam berbagai platform.

Analisis Pemecahan Masalah

1. Analisis Permasalahan

Salah satu tantangan utama dalam pengembangan aplikasi ini adalah banyaknya jumlah elemen yang terdapat dalam permainan Little Alchemy 2, yang mencapai lebih dari 700+ elemen dengan ribuan kombinasi recipe yang memungkinkan. Untuk memperoleh data ini secara akurat dan lengkap, dilakukan proses *scraping* dari situs wiki resmi Little Alchemy 2 yang menyimpan seluruh informasi kombinasi elemen. Hasil scraping kemudian disimpan dalam sebuah basis data dengan struktur yang terorganisasi, yaitu dengan memetakan setiap elemen ke daftar pasangan elemen pembentuknya. Struktur ini memungkinkan efisiensi dalam pencarian recipe serta memudahkan proses konversi ke bentuk graf.

Salah satu permasalahan teknis yang muncul adalah terjadinya *looping recipe*, yaitu kondisi di mana elemen A bisa membentuk elemen B, dan elemen B dapat kembali membentuk elemen A melalui jalur kombinasi berbeda. Jika tidak ditangani, hal ini dapat menyebabkan algoritma pencarian terjebak dalam siklus tak berujung. Untuk menghindari masalah ini, dilakukan penambahan mekanisme pencatatan simpul yang telah dikunjungi dalam setiap pencarian menggunakan struktur *visited set*. Dengan cara ini, algoritma dapat menghindari eksplorasi ulang terhadap elemen yang telah muncul dalam jalur yang sedang ditelusuri.

Permasalahan lain yang ditemukan adalah performa pencarian ketika diminta untuk menemukan banyak recipe (*multiple recipe mode*). Tanpa optimasi, pencarian ini memerlukan eksplorasi ruang kombinasi yang sangat besar dan waktu eksekusi yang lama. Untuk mengatasinya, diterapkan pendekatan *multithreading* pada sisi backend menggunakan bahasa Go. Setiap thread dapat secara paralel menelusuri jalur kombinasi yang berbeda, sehingga proses pencarian recipe dalam jumlah banyak dapat dilakukan secara lebih cepat dan efisien.

Selain itu, visualisasi hasil pencarian juga menjadi tantangan tersendiri. Menyusun jalur kombinasi elemen menjadi struktur pohon (*recipe tree*) yang mudah dipahami pengguna memerlukan perancangan format data dan tampilan yang tepat. Untuk mengatasi ini, dibangun skema representasi pohon secara hierarkis, di mana setiap node merepresentasikan sebuah elemen, dan cabangnya menunjukkan elemen pembentuknya. Visualisasi ini diimplementasikan

pada sisi frontend dengan pustaka diagram yang mendukung struktur tree, sehingga pengguna dapat dengan mudah menelusuri tahapan pembentukan suatu elemen dari akar hingga ke daun.

2. Pemetaan Elemen Permainan

Setiap elemen dalam permainan Little Alchemy 2 dapat direpresentasikan sebagai simpul (*node*) dalam sebuah graf, sedangkan setiap kombinasi antara dua elemen untuk membentuk elemen baru direpresentasikan sebagai sisi (*edge*) yang menghubungkan simpul-simpul tersebut. Dengan pendekatan ini, permainan dapat dipandang sebagai graf berarah, di mana arah dari sisi menunjukkan proses pembentukan dari dua elemen sumber menuju satu elemen hasil. Pendekatan graf ini memudahkan penerapan algoritma pencarian seperti BFS dan DFS untuk menelusuri jalur pembentukan elemen dari elemen dasar hingga elemen target.

Selain sebagai graf, data elemen dan kombinasinya juga dipetakan menjadi struktur pohon (*tree*) untuk keperluan visualisasi hasil pencarian recipe. Dalam struktur ini, simpul utama (root) mewakili elemen tujuan yang ingin dibentuk. Dua simpul anak (child) yang berada di bawahnya merupakan elemen yang harus dikombinasikan untuk membentuk simpul tersebut. Setiap simpul anak tersebut dapat memiliki anak-anak lagi, menggambarkan kombinasi yang dibutuhkan untuk membentuknya, dan seterusnya hingga simpul-simpul paling dasar yang merupakan elemen dasar permainan.

Informasi yang disimpan dalam setiap simpul pohon meliputi nama elemen dan indeks unik untuk membedakan tiap node secara internal. Data ini digunakan untuk dua tujuan utama: pertama, untuk menampilkan hierarki pembentukan elemen kepada pengguna secara visual; dan kedua, untuk memungkinkan frontend mengolah dan merender struktur pohon secara dinamis dan terstruktur. Representasi pohon ini memberikan pemahaman yang intuitif kepada pengguna mengenai tahapan-tahapan kombinasi yang diperlukan, terutama ketika sebuah elemen memiliki banyak lapisan kombinasi.

Dengan pemetaan seperti ini, aplikasi tidak hanya dapat menelusuri recipe dengan efisien melalui algoritma graf, tetapi juga menyajikan hasil pencarian tersebut dalam bentuk visual yang mudah dipahami. Hal ini memperkuat nilai edukatif aplikasi dengan menunjukkan secara

eksplisit bagaimana sebuah elemen terbentuk dari gabungan elemen-elemen sebelumnya secara bertingkat.

3. Fitur Fungsional dan Arsitektur Sistem

3.1 Penyelesaian Secara BFS

Pada penyelesaian masalah secara BFS (Breadth-First Search), pendekatan dilakukan dengan menjelajahi semua kemungkinan kombinasi elemen secara bertingkat, dimulai dari elemen dasar seperti Air, Fire, Water, dan Earth.

3.1.1 Single

Untuk kasus single/one recipe, BFS digunakan untuk mencari satu jalur yang bisa menghasilkan elemen target dari kombinasi dua elemen yang sah pada setiap langkahnya. Algoritma ini akan memprioritaskan jalur yang memiliki jumlah langkah paling sedikit, dengan cara memeriksa semua kombinasi pada level (tier) saat ini sebelum melanjutkan ke level (tier) berikutnya. Ketika elemen target ditemukan pertama kali, pencarian dihentikan dan jalur yang ditemukan akan direkonstruksi sebagai solusi.

3.1.2 Multiple

pada kasus multiple, BFS dimodifikasi untuk tidak hanya berhenti ketika satu jalur ditemukan, tetapi untuk terus mengeksplorasi seluruh kemungkinan jalur pembuatan elemen target, selama masih berada dalam batas tier yang valid dan batas jumlah jalur yang diizinkan. Dengan demikian, pendekatan ini memungkinkan pengguna untuk melihat berbagai cara berbeda dalam membentuk elemen yang sama, memperluas eksplorasi terhadap variasi kombinasi yang mungkin.

3.2 Penyelesaian Secara DFS

Pada penyelesaian masalah secara DFS (Depth-First Search), pendekatan dilakukan dengan menyusuri jalur pencarian sedalam mungkin sebelum kembali (*backtrack*) untuk mencoba jalur alternatif.

3.2.1 Single

Untuk kasus single/one recipe, DFS digunakan untuk menemukan satu jalur pembuatan elemen target dari elemen dasar, tanpa menjamin bahwa jalur tersebut merupakan yang tercepat atau paling efisien. Pencarian dilakukan secara rekursif, di mana setiap cabang eksplorasi akan ditelusuri hingga elemen target ditemukan atau tidak ada lagi jalur yang dapat dilalui. Begitu target ditemukan pertama kali, proses langsung dihentikan dan jalur tersebut dikembalikan sebagai solusi.

3.2.2 Multiple

Pada kasus multiple, DFS dimodifikasi agar dapat menelusuri semua kemungkinan jalur pembentukan elemen target yang valid hingga jumlah jalur yang ditemukan mencapai batas maksimum yang ditentukan. Pendekatan ini tetap menggunakan pencarian mendalam, namun tidak berhenti setelah satu solusi ditemukan, melainkan terus menjelajahi jalur-jalur lain yang mungkin ada. Hal ini memungkinkan analisis lebih dalam terhadap semua cara pembentukan suatu elemen. Akan tetapi, diperlukan pengecekan terhadap duplikasi jalur.

3.3 Strategi Bidirectional

Pada penyelesaian secara Bidirectional, pendekatan dilakukan dengan menjalankan dua pencarian secara simultan, yaitu satu dari elemen-elemen dasar (*forward search*) dan satu lagi dari elemen target menuju ke elemen dasar (*backward search*). Tujuannya adalah mempertemukan kedua pencarian di suatu titik tengah, sehingga ruang pencarian yang harus dijelajahi bisa jauh lebih kecil dibanding pencarian satu arah penuh. Bidirectional yang diterapkan pada tugas kali ini adalah bidirectional secara BFS dan hanya diimplementasi untuk single/one recipe. Proses akan berhenti begitu ditemukan elemen perantara yang berada di jalur dari elemen dasar ke target dan

memiliki rute valid dari kedua arah. Dengan menyambungkan jalur dari awal ke tengah dan dari tengah ke akhir, terbentuk satu solusi lengkap pembentukan target.

3.4 Arsitektur Program

Program dibagi menjadi 2 bagian yaitu frontend dan backend. Bagian frontend bertugas untuk menerima input yang diinginkan oleh user serta akan memvisualisasikan hasil tree yang dibuat. Frontend juga berfungsi sebagai validasi input sebelum nantinya dikirim ke backend. Selanjutnya, frontend bertanggung jawab untuk mengirimkan data ke backend yang nantinya akan diproses oleh backend dan ditampilkan hasilnya.

Bagian backend berfungsi untuk memproses input yang diterima oleh frontend. Terdapat 2 komponen penting yang ada pada backend yaitu algoritma yang pada tugas ini berisi algoritma BFS, DFS, dan Bidirectional serta API yang berfungsi sebagai penghubung frontend dan algoritma. Algoritma akan dijalankan sesuai pilihan yang diinginkan oleh user.

Alur dalam web ini adalah sebagai berikut. Pertama, user akan memasukan input yang dibutuhkan pada bagian frontend. Selanjutnya, API akan memproses input yang diinginkan oleh user. API akan memanggil algoritma yang dipilih oleh user lalu diproses di dalam algoritmanya. Setelah proses algoritma selesai, file JSON akan dibuat dan akan dipanggil kembali oleh API untuk selanjutnya divisualisasikan pada bagian frontend. Hasil yang keluar berupa visualisasi tree, banyaknya node yang dikunjungi, algoritma yang dipilih, dan banyaknya waktu yang dibutuhkan selama proses algoritma.

4. Contoh Ilustrasi Kasus

User memasukan bahan yang ingin dicari serta memilih algoritma dan mode yang diinginkan. Sebagai contoh, user ingin mencari Lake menggunakan algoritma BFS. Setelah input dimasukan, user menekan tombol search untuk mencari apa saja yang dibutuhkan untuk membuat Lake. Masukan dari pengguna akan dikirim ke backend melalui API. API akan memanggil fungsi BFS dan bahan-bahan untuk membuat Lake di dalam fungsi tersebut. Setelah

hasil ditemukan, API akan membuat file JSON lalu dikirimkan ke frontend untuk selanjutnya ditampilkan hasil tree visualisasi, banyak node yang dikunjungi, algoritma yang dipilih, dan banyaknya waktu pemrosesan. User juga dapat menggunakan tombol start animation untuk melihat bagaimana tree dibuat.

Implementasi dan Pengujian

1. Spesifikasi Teknis Program dan Struktur Proyek

    a. Struktur Proyek

    Proyek ini memiliki struktur direktori dan file sebagai berikut:

```
Tubes2-stimacino-banataino/
├── backend
│   └── api
│   ├── bfs
│       └── graph/
│       ├── result_BFS/
│       ├── result_multi_BFS/
│       ├── search
│           └── bfs.go
│           └── multiple_bfs.go
│       ├── .gitkeep
│       ├── go.mod
│       ├── main.go
│       ├── multi_main.go
│   ├── bidirectional
│       └── graph/
│       ├── result_Bidirectional/
│       ├── search
│           └── bidirectional.go
│       ├── .gitkeep
│       ├── go.mod
│       ├── main.go
│   ├── dfs
│       └── graph/
│       ├── result_DFS/
│       ├── result_multi_DFS/
│       ├── search
│           └── dfs.go
│           └── multiple_dfs.go
│       ├── .gitkeep
│       ├── go.mod
│       ├── main.go
│       ├── multi_main.go
│   ├── scraping
│       └── data_scraping
│       ├── .gitkeep
│       ├── go.mod
│       ├── go.sum
│       ├── main.go
│   ├── .gitkeep
```

```
├── docs
├── frontend
│   └── app/
│   ├── .gitkeep
└── README.md
```

b. Graph

Dalam folder graph ini terdapat dua file, yaitu loader.go dan type.go. type.go berisi struktur data yang akan digunakan dalam sistem pencarian elemen berbasis graf. Terdapat struktur Element yang merepresentasikan satu elemen lengkap dengan semua resepnya. Resep-resep ini berbentuk array dua dimensi string ([]string). Elemen-elemen ini dikelompokkan ke dalam struktur Tier, yaitu struktur yang menyimpan nama tier dan daftar elemennya. Seluruh data hasil scraping akan dimuat ke dalam struktur ScrapedData, yang merupakan *root* dari data JSON.

Selain itu, terdapat tipe Graph untuk map[string][][]string yang memetakan nama elemen ke daftar resep yang dapat membentuknya. Terdapat juga struktur TreeNode, yang menyimpan nama elemen, urutan node pencarian (NodeDiscovered), dan anak-anak node yang merepresentasikan elemen-elemen pembentuknya. Untuk menyusun hasil pencarian secara utuh, digunakan struktur TreeResult yang menyimpan pohon hasil (Tree), nama algoritma yang digunakan, durasi pencarian dan jumlah node yang dikunjungi. Untuk kasus pencarian multiple, digunakan struktur MultiTreeResult yang menyimpan banyak pohon dalam satu hasil pencarian.

loader.go berfungsi untuk memuat dan mengolah data resep elemen dari file JSON yang telah di-scraping. Fungsi LoadCatalog berfungsi untuk membaca file JSON dari path yang diberikan dan mendeserialisasi (unmarshal) isinya menjadi struktur ScrapedData yang kemudia merepresentasikan data hierarki elemen-elemen dalam permainan. Fungsi LoadRecipes berfungsi untuk membaca dan mengonversikan data tersebut menjadi struktur Graph. Terakhir, fungsi MapElementToTier menghasilkan map dari nama elemen ke indeks tiernya.

c. bfs.go

bfs.go merupakan file implementasi algoritma BFS untuk menemukan satu jalur pembuatan suatu elemen. Fungsi utamanya, yaitu BFS menerima parameter elemen target, struktur graf resep, dan pemetaan elemen ke tingkat tiernya. Tujuan dari fungsi ini adalah untuk mencari jalur dari elemen-elemen dasar ke elemen target (*starting element to target*). BFS melakukan inisialisasi dengan membuat struktur discovered untuk menyimpan node yang telah dikunjungi dalam struktur bentuk TreeNode. parentMap digunakan untuk menyimpan hubungan antara bahan pembentuk dengan setiap produk (elemen yang dihasilkan). Queue digunakan untuk menyimpan elemen-elemen yang akan dievaluasi secara BFS.

Proses BFS dilakukan dengan menelusuri elemen satu per satu dari antrian queue. Algoritma akan memeriksa seluruh produk yang dapat dibuat dari elemen yang sudah ada berdasarkan graf resep yang diterima di parameter BFS. Jika ditermukan resep yang dapat dibuat dari elemen-elemen pembentuk, dan tier bahan pembentuk lebih rendah dari produk (elemen yang dihasilkan), maka produk ini akan masuk ke dalam queue untuk dievalusasi berikutnya. Apabila produk yang didapat adalah target yang dicari, maka pencarian akan berhenti dan jalur pembentuk target akan dibangun dengan fungsi buildTree.

Fungsi buildTree akan menyusun pohon dari elemen target hingga ke elemen dasar berdasarkan parentMap. Setelah pohon terbentuk, fungsi setDiscoveredIndex akan digunakan untuk memetakan urutan penemuan node secara *reverse* (dimulai dari target dulu) yang berguna untuk fitur live update.

d. multiple_bfs.go

multiple_bfs.go merupakan implementasi dari fitur *multiple* BFS, yaitu pencarian BFS untuk menemukan semua kemungkinan rute pembuatan suatu elemen target berdasarkan kombinasi bahan-bahan yang tersedia. Fungsi utamanya, yaitu MultiBFS menerima parameter yang sama dengan BFS, tetapi dengan penambahan satu parameter lagi, yaitu maxRecipes yang merupakan batas maksimum rute yang ingin dicapai.

Algoritma dimulai dengan melakukan inisialisasi terhadap node-node dasar (*starting element*) dan memasukkan elemen-elemen ini ke dalam map existing yang menyimpan semua node TreeNode yang sudah terbentuk. Proses pencarian dilakukan secara bertahap berdasarkan tier elemen. Pada setiap iterasi, MultiBFS akan membangun kombinasi baru dari elemen yang sudah ada menggunakan daftar resep. Setiap kombinasi akan ditugaskan sebagai pekerjaan (levelTask) yang berisi nama produk, reseo bahan, dan node bahan kirinya dan kanannya. Algoritma menggunakan *worker pool* berbasis goroutine untuk memproses kombinasi secara paralel.

Sebelum membuat node baru, sistem akan mengecek apakah kombinasinya sama dengan yang sudah diproses sebelumnya atau tidak dengan menggunakan sync.Map (visitedCombo). Jika sebuah node yang baru terbentuk adalah elemen target, node tersebut akan disalin dengan fungsi deepCopyTree dan disimpan sebagai salah satu resep yang valid dalam foundRecipe. Untuk resep yang pertama ditemukan akan dijadikan sebagai resep utama dengan nilai NodeDiscoverednya tidak 0, sehingga akan diimplementasikan live update nantinya. Proses pencarian jalur akan diteruskan hingga tidak ada lagi kombinasi yang dapat diproses atau jumlah resep yang ditemukan sudah cukup.

e. dfs.go

dfs.go berisi implementasi dari DFS untuk menemukan satu jalur pembuatan untuk elemen target. Untuk struktur data yang digunakan tidak banyak berbeda dengan BFS, seperti Graph, TreeNode, discovered, hanya saja di DFS ini prosesnya dilakukan secara rekursif atau dengan stack. Algoritma DFS dimulai dengan meninjau elemen dasar yang tersedia sebagai titik awal pencarian. Fungsi rekursif dfs digunakan untuk mengunjungi setiap elemen yang mungkin dengan mengeksplorasi resep yang membentuk produk (elemen baru). DFS juga akan memeriksa apakah elemen sudah pernah dikunjungi atau tidak. Proses ini juga menerapkan *backtracking*, yaitu setelah mengeksplorasi jalur yang lebih dalam, pencarian akan mundur untuk mengeksplorasi jalur yang lain.

Setelah menemukan produk target, fungsi buildTree akan digunakan untuk membangun kembali pohon dari node target menuju ke elemen dasar. SetDiscoveredIndex akan memberikan urutan node yang dikunjungi secara *reverse* (dalam artian target akan yang elemen terakhir ditemukan akan diberi indeks 0)

f. dfs_multiple.go

File ini berisi implementasi dari algoritma multiple DFS, yaitu perluasan dari DFS yang dapat menemukan banyak resep untuk target. Fungsi utamanya, MultiDFS menerima parameter yang sama dengan DFS, hanya saja mendapat satu parameter tambahan, yaitu maxRecipe yang merupakan batas resep yang ingin dicapai. Struktur utama yang digunakan adalah map[string]Recipe dengan Recipe adalah struktur data yang menyimpan dua nama elemen pembentuk. Berbeda dengan tiga algoritma sebelumnya, MultiDFS melakukan DFS pertama kali dari elemen target melalui fungsi DfsFromTarget dan dengan rekursif akan menyusun rute untuk pembuatan elemen target hingga ke elemen dasar. Program ini juga menggunakan memoisasi untuk menghindari perhitungan ulang suatu elemen. Jika jalur utama berhasil ditemukan, akan disimpan sebagai primaryPath yang dapat divisualisasikan di fitur live update.

Selanjutnya, program akan mencari node-node yang memungkinkan memiliki alternatif resep melalui fungsi findElementsWithAlternatives. Fungsi ini menelusuri pohon hasil jalur utama dan mencatat elemen yang punya resep alternatif (yang berbeda dari yang digunakan di jalur utama dan tetap valid secara tier). Elemen-elemen ini menjadi kandidat untuk dieksplorasi ulang dalam fungsi exploreAlternatives, yang merupakan inti dari pencarian jalur alternPada tahap exploreAlternatives, dilakukan proses paralel menggunakan worker pool (jumlah worker disesuaikan dengan jumlah CPU) untuk mencoba mengganti resep dari elemen-elemen kandidat dengan resep alternatif. Setiap alternatif diperiksa validitasnya menggunakan fungsi validateAndRepairRecipe, yang menjamin bahwa jalur alternatif tetap sah dari elemen dasar dan semua komponennya memiliki tier yang lebih rendah dari produk yang dibentuk. Jika valid, pohon baru

dibangun dari map resep menggunakan buildTreeFromMap dan dimasukkan ke dalam daftar hasil. Untuk menghindari duplikasi, setiap jalur memiliki key unik yang di-*generate* dengan GenerateKey.

g. bidirectional.go

bidirectional.go berisi file implementasi dari strategi bidirectional secara BFS untuk menemukan jalur pembuatan elemen target. Bidirectional bekerja secara dua arah, yaitu dari elemen dasar (*forward*) dan dari elemen target (*backward*). Struktur utama yang digunakan terdiri dari map untuk mencatat kunjungan dari suatu node (forwardVisited, backwardVisited), hubungan antar elemen dengan pembentuknya (forwardParent, backwardParent), dan daftar elemen yang berada di setiap tier arah pencarian (forwardTier, backwardTier).

Bidirectional sebagai fungsi utamanya akan melakukan iterasi. Pada pencarian maju, akan dicoba untuk membentuk elemen baru dari kombinasi dua elemen yang sudah dikunjungi dan menyimpannya. Pada pencarian mundur, akan ditelusuri resep yang membentuk target. Jika ditemukan titik temu antara kedua arah, proses akan dihentikan dan hasil pencarian akan dibangun dengan fungsi buildTree dengan menggabungkan peta relasi dari kedua arah menggunakan mergeMaps. Untuk bidirectional, NodeDiscoverednya diatur bernilai 0 semua karena tidak ada fitur live update.

h. main.go dan multi_main.go

main.go dan multi_main.go merupakan file driver untuk menguji masing-masing algoritma yang sudah dibuat.

i. data_scraping

Folder ini berisi algoritma untuk scraping dan hasil scraping dalam bentuk JSON serta hasil scraping untuk SVG gambar elemen. Untuk algoritma scraping terdapat pada file scraper.go. Fungsi utamanya, ScrapeAll akan mengakses halaman utama dari website dan melakukan *parsing* dengan menggunakan pustakan goquery. Fungsi ini akan mengambil informasi nama resep, resep

pembentuknya, tier, dan tautan file SVG. Setiap elemen dikelomppokan berdasarkan tier. File SVG untuk masing-masing elemen akan diunduh dan disimpan secara lokal dalam floder dan seluruh informasi akan diserialisasi ke file JSON bernama scraped_data.json. Selain itu, program ini juga telah melakukan filter untuk beberapa elemen, seperti *special element* dan elemen-elemen yang ada di Myths dan Monster.

j. app

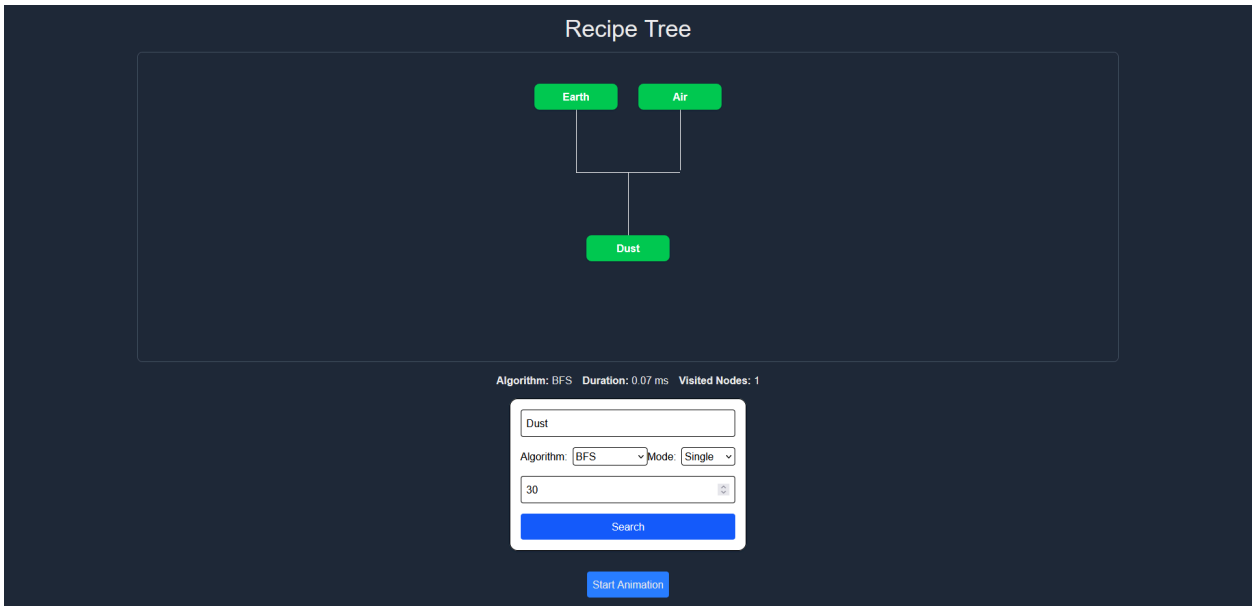
Struktur folder ini merupakan sebuah *frontend* yang dirancang untuk memisahkan logika, tampilan, dan konfigurasi agar lebih terorganisir dan mudah dipelihara. Folder utama seperti public/ digunakan untuk menyimpan aset statis seperti gambar dan ikon, sementara src/ menjadi inti dari pengembangan aplikasi. Di dalam src/, terdapat folder components/ yang berisi komponen UI yang dapat digunakan kembali, pages/ atau app/ (tergantung pada versi dan penggunaan Next.js) yang memuat halaman-halaman utama aplikasi, serta styles/ untuk menyimpan file CSS atau styling lainnya. Selain itu, file seperti package.json menyimpan daftar dependensi dan skrip penting untuk menjalankan proyek, dan .env digunakan untuk konfigurasi variabel lingkungan. Struktur ini bertujuan memisahkan tanggung jawab setiap bagian aplikasi sehingga pengembangan dapat dilakukan secara modular dan efisien.

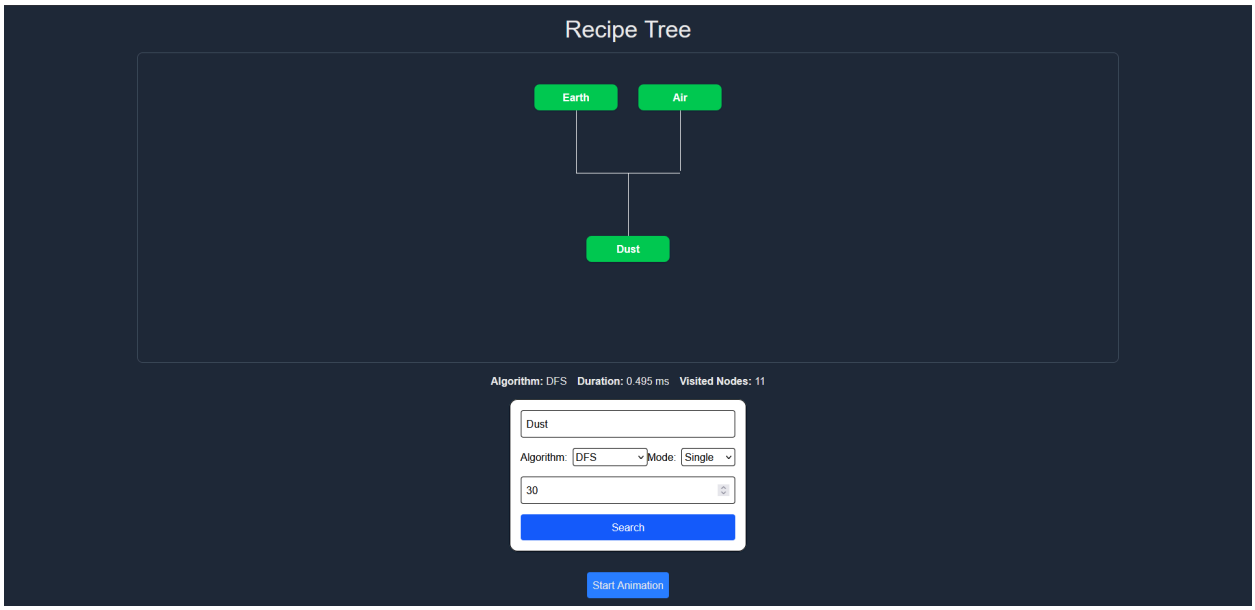
2. Pengujian

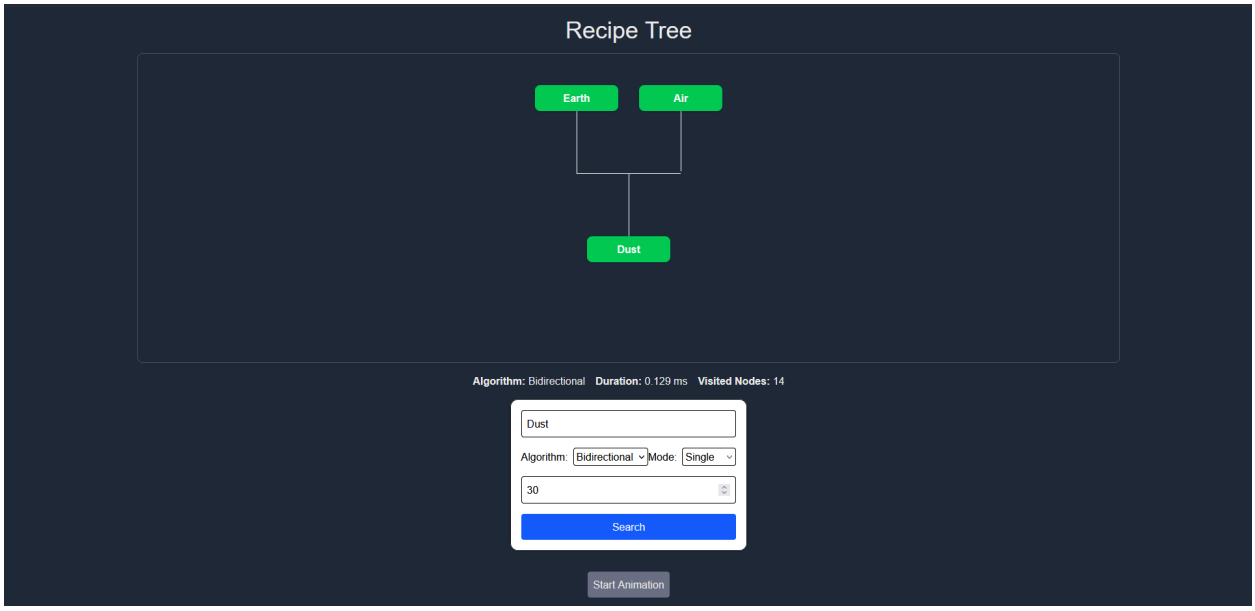
Untuk menjalankan program ini, user dapat masuk ke halaman berikut: https://tubes2-stimacino-banataino.vercel.app/. Setelah itu, user dapat memasukkan elemen target yang ingin dicari melalui bagian enter target. User dapat memilih algoritma yang diinginkan (BFS, DFS, atau Bidirectional). Selain itu, user juga dapat memilih mode yang diinginkan (single atau multiple, kebetulan untuk bidirectional tidak tersedia multiple). Jika user memilih single, user bisa mengabaikan bagian untuk memasukkan jumlah resep yang diinginkan. Jika user memilih multiple, user bisa memasukkan jumlah resep yang diinginkan. Selain itu, ketika sudah ada hasilnya, user dapat memilih ingin menampilkan resep ke berapa untuk mode

multiple. Terakhir, user dapat menggunakan fitur start animation untuk melihat live update dari proses pencarian (kecuali untuk bidirectional)

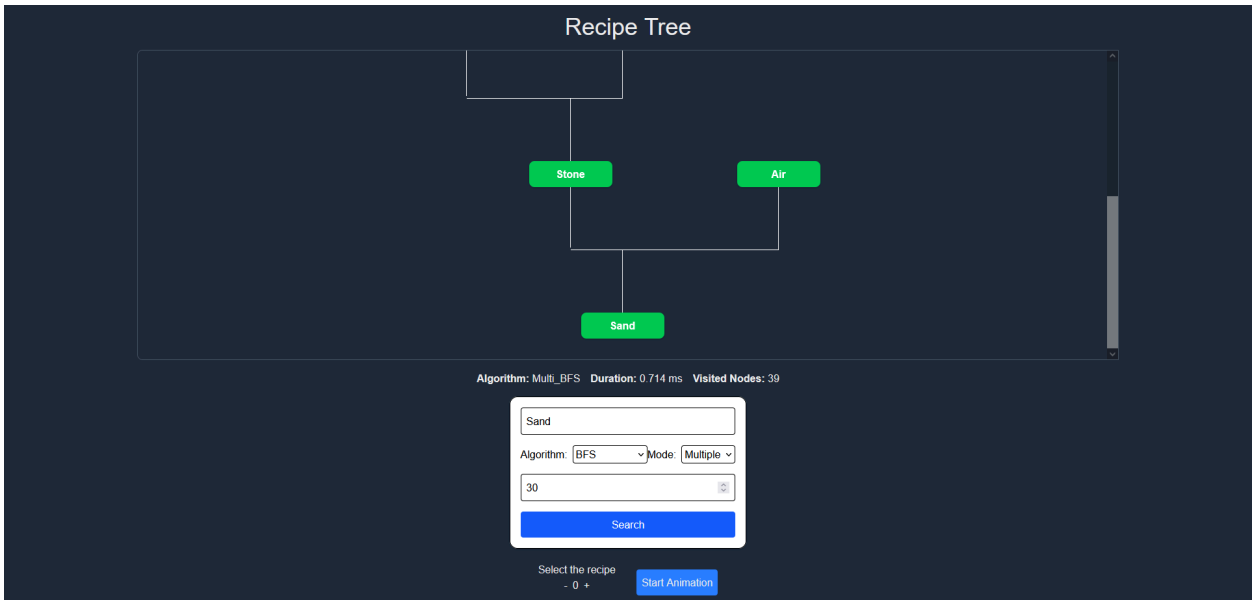
- **Percobaan untuk Elemen Dust**



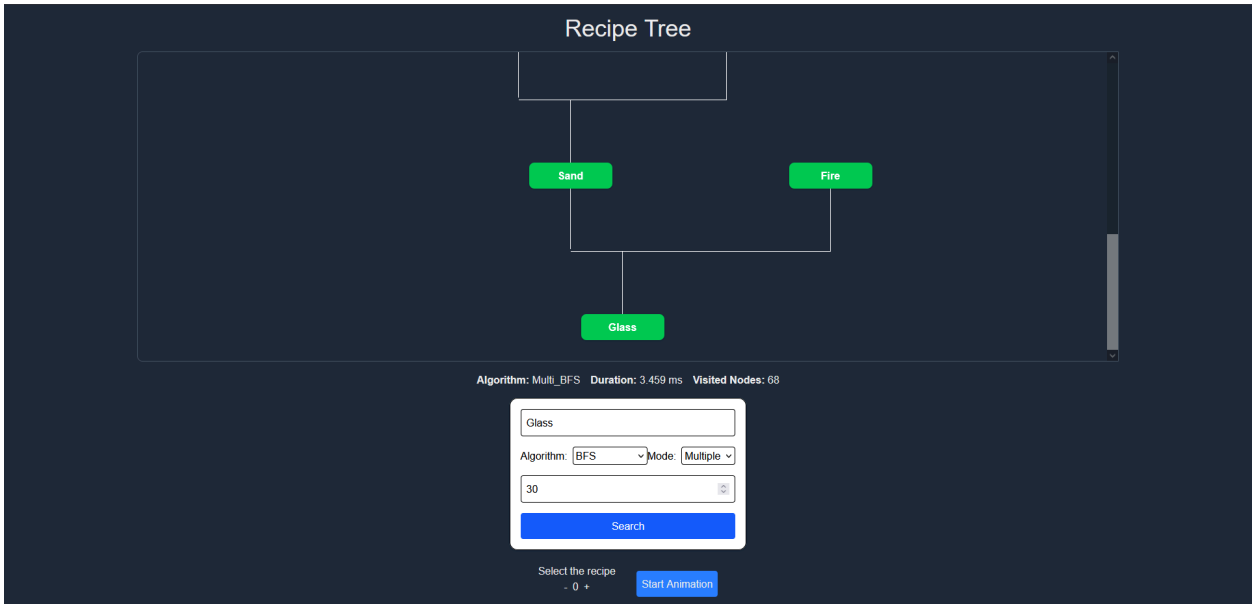


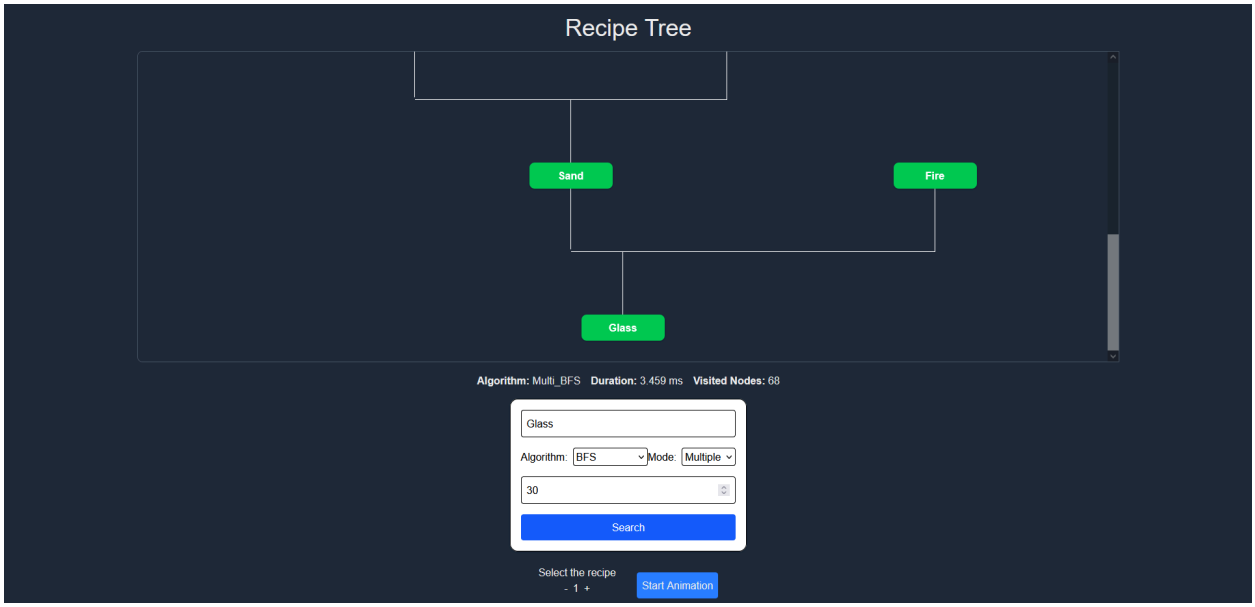


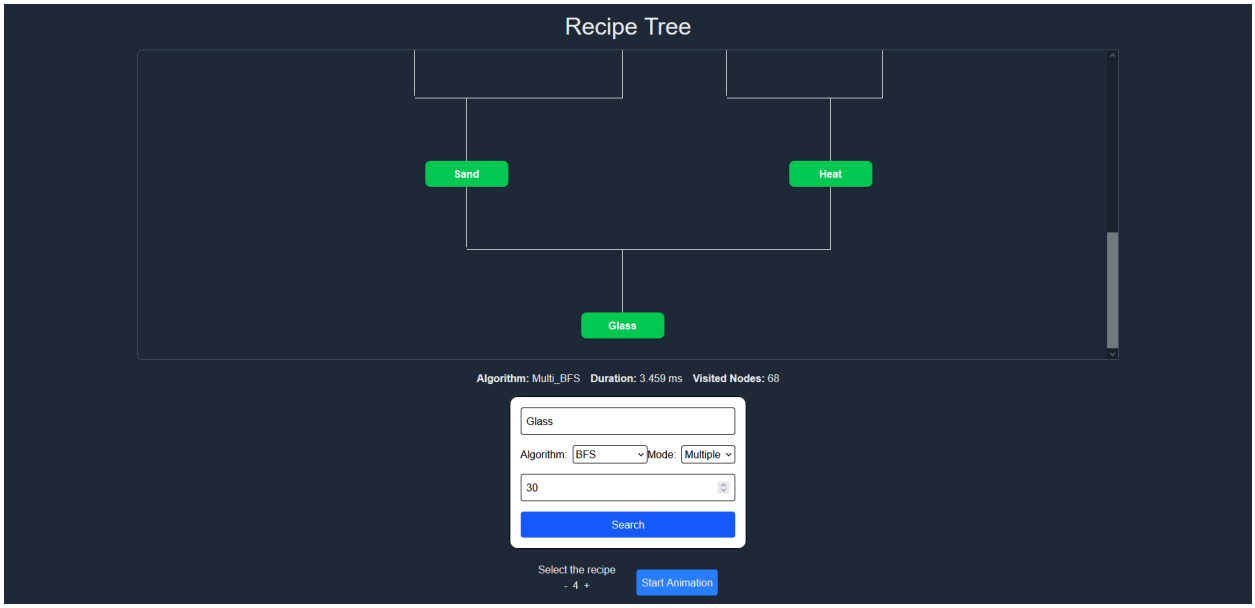
- **Percobaan untuk Elemen Sand**



- **Percobaan untuk Elemen Glass**


Recipe Tree
Sand
Fire
Glass
Algorithm: Multi_BFS Duration: 3.459 ms Visited Nodes: 68
Glass
Algorithm: BFS Mode: Multiple
30
Search
Select the recipe
- 0 +
Start Animation


Recipe Tree
Sand
Fire
Glass
Algorithm: Multi_BFS Duration: 3.459 ms Visited Nodes: 68
Glass
Algorithm: BFS Mode: Multiple
30
Search
Select the recipe
- 1 +
Start Animation



3. Analisis Hasil Pengujian

Algoritma BFS menunjukkan performa yang lebih unggul dalam menemukan jalur terpendek (*shortest-path*), terutama untuk mode *single*. BFS melakukan eksplorasi seluruh elemen berdasarkan tier, dimulai dari elemen dasar hingga mencapai elemen target. Akan tetapi, secara teoritis, kompleksitas waktu dan ruang dari BFS adalah O(b^d) dengan b adalah rata-rata banyak cabang (*branching factor*) dan d adalah kedalaman jalur menuju target. Pada mode multiple, BFS dimodifikasi untuk mengeksplorasi seluruh jalur alternatif hingga batas jumlah resep tertentu. Beban komputasi pada mode multiple meningkat signifikan karena BFS harus memperluas ruang pencarian secara eksponensial. Kompleksitas waktu untuk mode multiple ini secara teori adalah O(b^d * r) dengan e adalah jumlah maksimum jalur yang ingin dicapai.

Berbeda dengan BFS, DFS mampu mengefisienkan memori dengan lebih baik. DFS bekerja dengan menyusuri cabang hingga kedalaman maksimum yang diizinkan sebelum melakukan *backtrack*. Dalam mode *single*, DFS dapat menemukan solusi dengan lebih cepat apabila solusi berada pada cabang awal-awal yang dieksplorasi. Akan tetapi, DFS tidak menjamin optimalisasi jalur. Kompleksitas waktu dari DFS sama seperti BFS, yaitu O(b^d), sedangkan kompleksitas ruangnya adalah O(d). Dalam mode multiple, DFS dapat menghasilkan banyak jalur tanpa menyimpan banyak simpul sekaligus. Akan tetapi, jalur-jalur yang ditemukan cenderung lebih panjang dan kurang optimal dibandingkan dengan multiple BFS.

Terakhir, Strategi Bidirectional merupakan pendekatan yang paling optimal dibandingkan dua lainnya (khusus untuk single, karena di percobaan hanya dilakukan pada single). Strategi ini bekerja dengan menggabungkan dua arah pencarian, yaitu dari elemen dasar ke target (*forward*) dan dari target ke elemen dasar (*backward*). Strategi ini dapat mengurangi waktu yang diperlukan untuk pencarian. Secara teoritis, pendekatan ini memiliki kompleksitas waktu dan ruang, yaitu O(b ^ (d / 2)).

Kesimpulan dan Saran

Dalam pengerjaan proyek ini, kami berhasil membangun sistem yang dapat memvisualisasikan pohon hasil dari proses pencarian resep berdasarkan algoritma tertentu. Selain menyelesaikan fitur utama, kami juga berhasil mengimplementasikan fitur bonus yang diminta, seperti pengolahan berbagai mode input dan visualisasi hasil dalam bentuk tree secara interaktif. Proyek dapat berjalan dengan baik sesuai dengan spesifikasi yang ditentukan.

Selama proses pengembangan, terdapat beberapa tantangan teknis yang kami temui. Salah satu kendala utama adalah terjadinya *infinite recursion* saat proses penelusuran elemen karena node anak yang sudah pernah ditemukan tetap diproses ulang. Masalah ini dapat diatasi dengan menerapkan strategi *early stopping* untuk menghentikan proses saat node yang sama terdeteksi kembali. Selain itu, kami juga mengalami kendala dalam visualisasi tree yang tidak seimbang secara struktur. Permasalahan ini diselesaikan dengan cara menghitung ukuran (size) dari setiap subtree dan menempatkan node secara dinamis agar keseluruhan tampilan pohon menjadi proporsional dan mudah dipahami.

Sebagai saran pengembangan ke depannya, tampilan antarmuka pengguna (frontend) dapat ditingkatkan agar lebih menarik dan intuitif. Penanganan kesalahan (error handling) juga masih bisa diperbaiki agar sistem lebih robust terhadap input atau kondisi tak terduga. Visualisasi tree saat ini masih memiliki keterbatasan dalam hal interaktivitas, seperti fitur *zoom in* dan *zoom out* yang belum tersedia karena keterbatasan dari implementasi CSS saat ini. Selain itu, penambahan fitur *autofill* dan *autocomplete* saat pengguna mengetik elemen target akan sangat membantu dalam meningkatkan kenyamanan dan efisiensi penggunaan sistem.

Lampiran

Tautan GitHub

Tautan Website Projek

Tautan Video Demo

Tabel Ketercapaian

No	Poin	Ya	Tidak
1	Aplikasi dapat dijalankan.	✅	
2	Aplikasi dapat memperoleh data *recipe* melalui scraping.	✅	
3	Algoritma *Depth First Search* dan *Breadth First Search* dapat menemukan recipe elemen dengan benar.	✅	
4	Aplikasi dapat menampilkan visualisasi recipe elemen yang dicari sesuai dengan spesifikasi.	✅	
5	Aplikasi mengimplementasikan multithreading.	✅	
6	Membuat laporan sesuai dengan spesifikasi.	✅	
7	Membuat bonus video dan diunggah pada Youtube.	✅	
8	Membuat bonus algoritma pencarian *Bidirectional*.	✅	
9	Membuat bonus *Live Update*.	✅	
10	Aplikasi di-*containerize* dengan Docker.	✅	
11	Aplikasi di-*deploy* dan dapat diakses melalui internet.	✅	

Daftar Pustaka

Munir, R., & Maulidevi, N. U. 2025. Algoritma BFS/DFS (Bagian 1). Diakses melalui https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf

Munir, R., & Maulidevi, N. U. 2025. Algoritma BFS/DFS (Bagian 2). Diakses melalui https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-(2025)-Bagian2.pdf