

LAPORAN TUGAS BESAR 2

Strategi Algoritma



Kelompok 29

FireBoyWaterGirl

Anggota Kelompok:

Muhammad Rusmin Nurwadin (13523068)

Aryo Bama Wiratama (13523088)

Reza Ahmad Syarif (13523119)

Program Studi Teknik Informatika
Institut Teknologi Bandung

DAFTAR ISI

1	DESKRIPSI TUGAS	5
2	LANDASAN TEORI	7
2.1	Graf	7
2.2	Breadth First Search	8
2.3	Depth First Search	9
3	ANALISIS PEMECAHAN MASALAH	11
3.1	Langkah-Langkah Pemecahan Masalah.....	11
3.2	Pemetaan Masalah Menjadi Elemen-Elemen Algoritma DFS dan BFS	11
3.2.1	Depth-First Search (DFS).....	11
3.2.2	Breadth-First Search (BFS)	11
3.3	Fitur Fungsionalitas dan Arsitektur Web.....	12
3.3.1	Fitur Fungsionalitas.....	12
3.3.2	Arsitektur Web	13
3.4	Ilustrasi Kasus	14
4	IMPLEMENTASI DAN PENGUJIAN	16
4.1	Spesifikasi Teknis Program	16
4.2	Tata Cara Penggunaan Program	22
4.3	Hasil Pengujian.....	22
4.4	Analisis Hasil Pengujian	26
5	KESIMPULAN	27
5.1	Kesimpulan	27
5.2	Saran.....	27
5.3	Komentar	27
5.4	Refleksi	27
6	LAMPIRAN	29
6.1	Tautan Repository Github.....	29
6.2	Tautan Video	29
7	DAFTAR PUSTAKA.....	29

DAFTAR GAMBAR

Gambar 1.1 Little Alchemy 2 (sumber: https://www.thegamer.com).....	5
Gambar 1.2 Elemen dasar pada Little Alchemy 2 (sumber: https://www.thegamer.com)	5
Gambar 2.1 Pohon Graf BFS (sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir)	9
Gambar 2.2 Pohon Graf DFS (sumber: https://www.trivusi.web.id/2022/05/apa-itu-algoritma-depth-first-search.html)	10

1 DESKRIPSI TUGAS



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

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, mahasiswa diminta 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.



Gambar 1.2
Elemen dasar pada Little Alchemy 2
(sumber: <https://www.thegamer.com>)

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.

2 LANDASAN TEORI

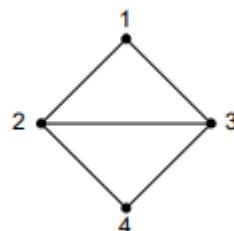
2.1 Graf

Graf adalah struktur diskrit yang terdiri dari simpul (*vertex*) dan sisi (*edge*) yang menghubungkan simpul-simpul tersebut. Secara formal, sebuah graf G didefinisikan sebagai $G = (V, E)$. Dalam hal ini, V merupakan himpunan tak-kosong dari simpul-simpul $\{v_1, v_2, \dots, v_n\}$, sedangkan E merupakan himpunan sisi yang menghubungkan pasangan simpul dalam V. Misalnya, pasangan simpul $\{(v_1, v_2), (v_2, v_3), \dots, (v_m, v_n)\}$ yang akan dinyatakan dengan himpunan sisi $\{e_1, e_2, \dots, e_n\}$.

Graf dapat diklasifikasikan berdasarkan beberapa kategori antara lain ada tidaknya gelang atau sisi ganda dan orientasi arah pada graf. Berdasarkan ada tidaknya gelang atau sisi ganda, graf dapat digolongkan sebagai berikut.

1. Graf Sederhana

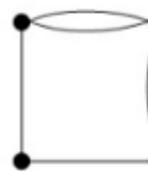
Graf yang tidak memiliki sisi ganda maupun gelang.



2. Graf Tak-Sederhana

Graf yang mengandung sisi ganda atau gelang. Graf ini dapat dibedakan lagi menjadi:

- Graf ganda atau multi-graph yaitu graf yang memiliki sisi ganda.



- Graf semu atau pseudo-graph yaitu graf yang mengandung sisi gelang.



Graf juga memiliki beberapa terminologi. Terminologi tersebut antara lain sebagai berikut.

1. Ketetanggaan (Adjacency)

Dua simpul dikatakan bertetangga jika keduanya terhubung langsung oleh sebuah sisi dalam graf.

2. Bersisian (Incidence)

- Sisi e dikatakan bersisian dengan simpul u dan v jika e menghubungkan kedua simpul tersebut. Dalam graf berarah, arah dari sisi menentukan bersisiannya.
3. Simpul Terpencil (Isolated Vertex)
Simpul yang tidak memiliki sisi yang bersisian dengannya disebut simpul terpencil.
 4. Graf Kosong (Null Graph)
Graf kosong adalah graf yang himpunan sisinya kosong, tetapi memiliki simpul.
 5. Derajat (Degree)
Derajat dari simpul adalah jumlah sisi yang bersisian dengan simpul tersebut. Dalam graf berarah, ada dua jenis derajat:
 - Derajat Masuk (In-degree): Jumlah sisi yang masuk ke simpul.
 - Derajat Keluar (Out-degree): Jumlah sisi yang keluar dari simpul.
 6. Lintasan (Path)
Lintasan adalah urutan simpul-simpul yang saling terhubung oleh sisi-sisi dalam graf, tanpa ada pengulangan sisi. Pada lintasan, simpul dapat diulang hanya jika tidak melanggar definisi, tetapi sisi tidak boleh diulang.
 7. Siklus atau Sirkuit
Siklus adalah lintasan yang dimulai dan diakhiri pada simpul yang sama tanpa mengulang sisi. Dalam graf takberarah, ini sering disebut siklus, sedangkan dalam graf berarah disebut sirkuit.
 8. Keterhubungan
Suatu graf dikatakan terhubung apabila terdapat lintasan yang menghubungkan setiap pasangan simpul. Graf dikatakan tidak terhubung jika terdapat setidaknya satu simpul yang tidak dapat dijangkau dari simpul lainnya.
 9. Upagraf (Subgraph) dan Komplemen Upagraf
Upagraf adalah sebuah graf yang himpunan dari simpul dan sisinya barunya merupakan himpunan bagian dari simpul dan sisi graf asal. Sedangkan, komplemen upagraf adalah graf yang berisi semua simpul dari graf asal tetapi memiliki sisi-sisi yang tidak ada dalam upagraf tersebut.
 10. Upagraf Merentang (Spanning Subgraph)
Upagraf yang mencakup semua simpul dari graf asal tetapi mungkin hanya memiliki sebagian sisi.
 11. Cut-Set
Cut-set adalah himpunan sisi yang, jika dihapus, menyebabkan graf menjadi tidak terhubung.
 12. Graf Berbobot
Graf berbobot adalah graf yang setiap sisi memiliki nilai atau bobot tertentu yang merepresentasikan jarak, biaya, atau parameter lainnya.

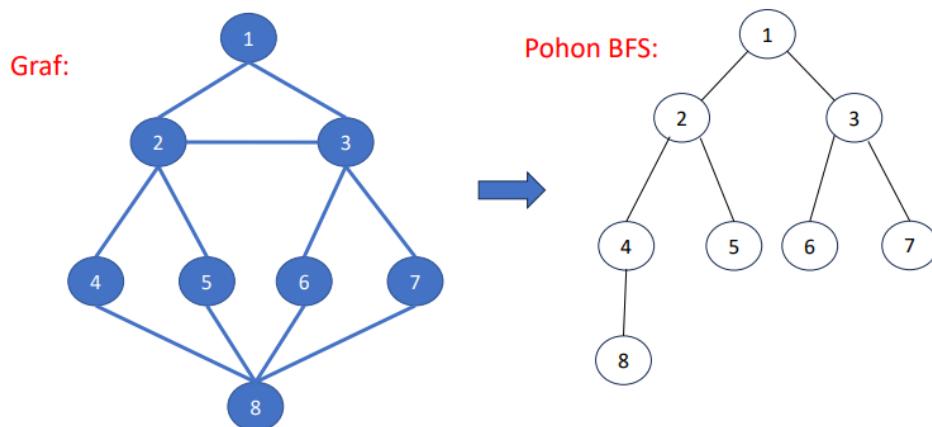
2.2 Breadth First Search

Algoritma Breadth First Search (BFS) adalah algoritma traversal graf yang secara sistematis menjelajahi graf dengan mengunjungi semua simpul pada level tertentu sebelum beralih ke level berikutnya. Dimulai dari simpul awal, memasukkannya ke dalam antrian, dan menandainya sebagai telah dikunjungi. Kemudian, mengeluarkan simpul dari antrian, mengunjunginya, dan memasukkan semua tetangganya yang belum dikunjungi ke dalam antrian. Proses ini berlanjut hingga antrian kosong.

Algoritma BFS melakukan transversal dari simpul v. Berikut adalah algoritma BFS dalam menelusuri suatu graf.

1. Kunjungi simpul v.
2. Kunjungi semua simpul yang bertetangga dengan simpul v terlebih dahulu.
3. Kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang tadi dikunjungi, demikian seterusnya.

Traversal graf secara BFS dapat digambarkan sebagai pohon BFS berikut dengan nomor sebagai urutan kunjungan.



Gambar 2.1
Pohon Graf BFS
(sumber: <https://informatika.stei.itb.ac.id/~rinaldi.munir>)

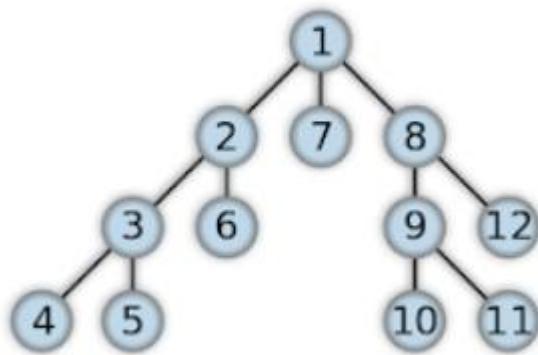
2.3 Depth First Search

Algoritma Depth First Search (DFS) adalah suatu metode pencarian pada sebuah tree/pohon dengan menelusuri satu cabang sebuah tree sampai menemukan solusi. Pencarian dilakukan pada satu node dalam setiap level dari yang paling kiri dan dilanjutkan pada node sebelah kanan. Jika solusi ditemukan maka tidak diperlukan proses backtracking yaitu penelusuran balik untuk mendapatkan jalur yang diinginkan.

Sama halnya dengan algoritma BFS, algoritma DFS memulai traversal dari simpul v. Berikut adalah algoritma DFS dalam menelusuri suatu graf.

1. Kunjungi simpul v.
2. Kunjungi simpul w yang bertetangga dengan simpul v.
3. Ulangi DFS mulai dari simpul w.
4. Ketika mencapai simpul u sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian diruntut-balik (backtrack) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul w yang belum dikunjungi.
5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.

Traversal graf secara DFS dapat digambarkan sebagai pohon BFS berikut dengan nomor sebagai urutan kunjungan.



Gambar 2.2
Pohon Graf DFS

(sumber: <https://www.trivusi.web.id/2022/05/apa-itu-algoritma-depth-first-search.html>)

3 ANALISIS PEMECAHAN MASALAH

3.1 Langkah-Langkah Pemecahan Masalah

1. Hal pertama yang dilakukan adalah melakukan identifikasi masalah untuk menemukan keinginan pengguna. Dari hasil identifikasi didapatkan bahwa pengguna ingin menemukan bagaimana cara membuat elemen tertentu (*targetElement*) dari kombinasi elemen dasar atau elemen lainnya. Algoritma DFS (*Depth-First Search*) atau BFS (*Breadth-First Search*) akan digunakan untuk menemukan jalur dari elemen dasar ke elemen target.
2. Kemudian, dilakukan *scrapping* data pada website [https://little-alchemy.fandom.com/wiki/Elements_\(Little_Alchemy_2\)](https://little-alchemy.fandom.com/wiki/Elements_(Little_Alchemy_2)) untuk mendapatkan berbagai macam element yang ada beserta dengan kombinasi resep yang dapat menghasilkan elemen tersebut.
3. Pengguna memasukkan elemen target dan memilih algoritma pencarian (DFS atau BFS). Algoritma pencarian akan diterapkan pada struktur pohon (kombinasi elemen yang bisa dibuat) untuk menemukan cara resep dari elemen target.
4. Algoritma DFS atau BFS diterapkan pada struktur pohon yang ada di backend untuk mencari solusi. Backend akan mengembalikan hasil dalam bentuk pohon (*tree*) yang menggambarkan bagaimana elemen target bisa dibentuk dalam bentuk json.
5. Hasil pencarian dari backend akan ditampilkan dalam bentuk pohon di frontend untuk memberi gambaran tentang hubungan antara elemen. Frontend juga akan memungkinkan fitur-fitur tambahan seperti pustaka elemen, *multiple recipe*, *live update*, dan *history search*.

3.2 Pemetaan Masalah Menjadi Elemen-Elemen Algoritma DFS dan BFS

Dalam aplikasi ini, pohon dari resep akan dipetakan menggunakan algoritma DFS atau BFS untuk menemukan jalur antara elemen dasar dan elemen target.

3.2.1 Depth-First Search (DFS)

Algoritma DFS akan melakukan traversal berdasarkan kedalaman. DFS memulai pencarian dari elemen dasar dan mengikuti satu jalur hingga mencapai leaf node atau elemen target, baru kemudian kembali untuk mencari jalur lain. Berikut adalah proses dari algoritma DFS.

1. Mulai dari node awal (elemen dasar).
2. Telusuri satu jalur hingga kedalaman maksimal yaitu hingga mencapai elemen dasar atau node leaf.
3. Setelah mencapai leaf node, kembali (*backtrack*) dan coba jalur lain untuk menemukan kombinasi elemen untuk membentuk elemen target.
4. DFS tidak menjamin jalur terpendek, tapi akan menemukan salah satu jalur ke target.

3.2.2 Breadth-First Search (BFS)

Algoritma BFS akan melakukan tranversal berdasarkan tingkat (level): BFS akan memulai pencarian dari elemen dasar dan menelusuri setiap level secara berurutan

(level by level) hingga mencapai elemen target. Berikut adalah proses dari algoritma BFS.

1. Mulai dari node awal (elemen dasar).
2. Kunjungi node yang terhubung langsung dengan node awal (*child nodes*).
3. Kunjungi semua anak node pada level pertama sebelum beralih ke level kedua, dan seterusnya.
4. BFS akan menemukan jalur terpendek dalam pohon (apabila ada beberapa jalur menuju target).

Pemetaan Masalah Pada Aplikasi:

1. Elemen: Setiap elemen (misalnya Mud, Fire, Brick) akan menjadi node dalam pohon.
2. *Edges*: Hubungan antara elemen yang bisa membentuk elemen lainnya (misalnya Mud + Fire menghasilkan Brick) akan menjadi *edges* yang menghubungkan dua node.
3. *Root*: Elemen target yang ingin dicari akan menjadi root atau tujuan akhir dari pencarian.

3.3 Fitur Fungsionalitas dan Arsitektur Web

Aplikasi web ini dibangun dengan berbagai fitur fungsional untuk memudahkan pengguna dalam menemukan resep alkimia berdasarkan elemen yang ada.

3.3.1 Fitur Fungsionalitas

1. Pencarian Elemen

Fitur pencarian elemen memungkinkan pengguna dapat memasukkan elemen target yang ingin dicari resepnya. Pengguna dapat memilih algoritma pencarian, baik Breadth-First Search (BFS) atau Depth-First Search (DFS), yang diinginkan untuk menemukan resep yang sesuai.

2. Pustaka Elemen

Fitur pustaka elemen memungkinkan pengguna dapat melihat berbagai elemen yang terdapat pada aplikasi web ini. Fitur ini akan menampilkan nama dan gambar masing-masing elemen. Dengan fitur ini, diharapkan pengguna nantinya bisa mengetahui berbagai elemen yang ada dalam aplikasi web ini dan bisa mencari tahu resep untuk membuat elemen yang ingin diketahui.

3. Visualisasi Graf

Fitur visualisasi graf memungkinkan pengguna dapat mengetahui berbagai macam kombinasi yang dilakukan untuk mendapatkan elemen target. Fitur ini akan menampilkan pohon atau graf resep yang menunjukkan hubungan antar elemen. Visualisasi ini diharapkan mampu memudahkan pengguna dalam mengetahui kombinasi yang dibuat sehingga mencapai elemen target.

4. Multiple Recipe

Fitur Multiple Recipe memungkinkan pengguna dapat melihat lebih dari satu resep dengan batasan jumlah resep yang ditampilkan bisa diatur oleh pengguna. Misalnya, jika pengguna memilih “2 resep”, aplikasi akan mengembalikan dua kombinasi resep berbeda untuk membuat Wave. Fitur ini didukung dengan fitur Pagination agar setiap kombinasi resep memiliki halaman yang berbeda sehingga

pengguna dapat dengan mudah memahami dan melihat perbedaan resep yang dihasilkan.

5. History Search

Fitur History Search memungkinkan aplikasi web untuk menyimpan riwayat pencarian yang pernah dicari oleh pengguna. Dengan fitur ini, pengguna dapat mengetahui riwayat pencarian yang pernah dilakukan dan dapat kembali melihat resep dari riwayat pencarian elemen tersebut.

6. Live Update

Fitur Live Update memungkinkan pengguna dapat melihat proses pencarian resep suatu elemen secara *real-time*. Fitur ini akan menampilkan visualisasi progres pencarian elemen secara bertahap, sehingga memungkinkan pengguna untuk melihat langkah-langkah pencarian.

3.3.2 Arsitektur Web

1. Frontend

Arsitektur web bagian Frontend dibangun dengan menggunakan React untuk UI yang dinamis. Dengan React, setiap elemen UI (seperti tombol, input form, dan hasil pencarian) dibangun sebagai komponen yang terpisah. Komponen-komponen ini bersifat modular dan dapat di-reuse, yang membuat pengelolaan dan pemeliharaan aplikasi menjadi lebih mudah. Selain itu, React memungkinkan pembaruan state dengan efisien, sehingga hanya bagian yang terpengaruh yang diperbarui, tanpa mempengaruhi keseluruhan halaman.

Kemudian, visualisasi pohon untuk memetakan kombinasi resep menggunakan library react-d3-tree. Library tersebut dipilih karena tergolong sederhana dan mudah untuk mengaplikasikan visualisasi pohon yang interaktif. Library ini nantinya akan menyediakan tampilan pohon resep yang hierarkis, memungkinkan pengguna untuk memahami langkah-langkah dalam pembuatan elemen target. Selain itu, library tersebut juga menggambarkan hubungan parent-child antara elemen yang digunakan untuk membuat elemen target, memudahkan visualisasi alur resep dari elemen dasar hingga target.

Frontend memanfaatkan hooks seperti `useRecipeSearch` untuk mengatur state pencarian dan komunikasi dengan backend. Hooks ini akan mengelola state pencarian seperti `loading`, `error`, dan `results`. Frontend juga memanfaatkan `React.lazy` untuk pemuatan komponen secara dinamis, seperti yang digunakan pada component `RecipeTree` dalam proses visualisasi pohon.

Terdapat juga *service API* yang berisi beberapa fungsi berfungsi untuk mengakses data dari backend melalui API, mengirimkan permintaan, dan memproses respons yang diterima. API ini berfokus pada pencarian resep alkimia dan interaksi dengan backend server untuk mendapatkan data resep dan elemen-elemen dasar yang digunakan dalam aplikasi. Fungsi `searchRecipe` akan mengirim permintaan ke API untuk mencari resep berdasarkan parameter: `targetElement`, `algorithm`, dan `n` yang merupakan jumlah pohon merentang. Kemudian, akan disimpan data yang dikembalikan berupa kombinasi elemen (combos) yang dibutuhkan untuk membuat elemen target dan statistik seperti waktu pencarian dan jumlah node yang dikunjungi.

2. Backend

Backend aplikasi ini dibangun menggunakan Go sebagai bahasa pemrograman dan Gorilla Mux untuk menangani routing API. Backend menggunakan Gorilla/Mux untuk melakukan routing HTTP, yang memungkinkan penanganan URL yang lebih kompleks dan mendukung berbagai jenis permintaan HTTP (GET, POST, dll). Backend ini mengimplementasikan berbagai algoritma pencarian seperti DFS dan BFS untuk menemukan kombinasi elemen yang dapat digunakan untuk membuat elemen target. Selain terdapat algoritma BFS dan DFS terdapat juga algoritma untuk melakukan scrapping data pada [https://little-alchemy.fandom.com/wiki/Elements_\(Little_Alchemy_2\)](https://little-alchemy.fandom.com/wiki/Elements_(Little_Alchemy_2)).

Backend juga menggunakan multithreading untuk melakukan pencarian secara paralel menggunakan goroutines. Penggunaan goroutines dalam algoritma BFS memungkinkan aplikasi untuk melakukan pencarian secara paralel, meningkatkan kecepatan dan efisiensi. sync.Mutex digunakan untuk menjaga integritas data bersama, sementara sync.WaitGroup memastikan bahwa semua goroutines selesai sebelum melanjutkan proses lebih lanjut. Channel digunakan untuk memberikan pembaruan progres secara real-time kepada pengguna jika diperlukan. Secara keseluruhan, multithreading dengan goroutines sangat penting dalam mempercepat pencarian resep dalam graf besar yang memiliki banyak elemen dan kombinasi. Proses yang sama juga bisa diterapkan pada algoritma DFS, di mana goroutines akan digunakan untuk melakukan pencarian dalam graf secara paralel, memaksimalkan pemanfaatan prosesor dan mengurangi waktu tunggu pengguna.

3.4 Ilustrasi Kasus

Misalkan pengguna ingin mencari resep membuat Stone. Akan dilakukan pencarian menggunakan algoritma BFS dan DFS.

- Algoritma BFS
 1. Stone adalah elemen target yang ingin dicari. Algoritma dimulai dengan elemen target ini.
 2. Backend kemudian akan mulai mencari kombinasi elemen yang akan menghasilkan Stone.
 3. Pada level pertama, Stone memiliki dua elemen anak yaitu Earth dan Pressure.
 4. BFS kemudian akan menjelajahi elemen-elemen pada level pertama ini. Stone dihasilkan dari kombinasi antara Earth dan Pressure, yang kemudian akan lanjut ke level berikutnya.
 5. Untuk Earth, dapat dilihat bahwa itu adalah elemen dasar, jadi Earth ditandai sebagai leaf Untuk Pressure, dapat dilihat bahwa Pressure terdiri dari dua elemen anak yaitu Air dan Air. Ini adalah kombinasi elemen dasar.

- Algoritma DFS
 1. Stone adalah elemen target yang ingin dicari. Algoritma dimulai dengan elemen target ini.
 2. Dari Stone, DFS akan memilih salah satu anak dan menelusuri cabang tersebut sepenuhnya. Dalam hal ini, anggap DFS bisa memulai dengan mengeksplorasi Earth.
 3. Earth adalah elemen dasar (leaf), jadi DFS akan kembali ke Stone setelah mencapai Earth.
 4. Setelah mengeksplorasi cabang pertama, DFS akan mengeksplorasi cabang kedua yaitu Pressure. Dari Pressure, DFS akan menelusuri lebih dalam ke Air dan Air.
 5. Begitu DFS mencapai daun (seperti Air), algoritma akan kembali ke node Pressure dan kemudian kembali ke Stone.
 6. DFS akan menyelesaikan pencarian setelah menelusuri seluruh cabang pohon.

4 IMPLEMENTASI DAN PENGUJIAN

4.1 Spesifikasi Teknis Program

1. Struktur Data

- Elemen

```
type Element struct{
    Name string `json:"name"`
    Recipes []IngredientPair `json:"recipes"`
    Tier int     `json:"tier"`
}
```

Digunakan untuk menyimpan elemen serta recipe yang dibutuhkan untuk membuatnya dari hasil scraping

- RecipeGraph

```
type RecipeGraph struct {
    Graph map[string]Element
}
```

Digunakan untuk menyimpan kumpulan elemen sehingga membentuk suatu graph

- Combo

```
type Combo struct{
    ID      int      `json:"id"`
    ParentId int     `json:"parentId"`
    Inputs  []string `json:"inputs"`
    Output  string   `json:"output"`
}
```

Digunakan pada proses BFS dan DFS combo akan dibentuk elemen yang telah dikunjungi

2. Fungsi dan Prosedur

Berikut adalah beberapa fungsi dan prosedur yang terdapat pada Backend dan berhubungan langsung dengan algoritma BFS dan DFS.

- BFS

```

•   func GetRecipeBFS(graph *types.RecipeGraph, target string, combos
•     *[]types.Combo, nRecipe int, progressChan chan types.Combo, isLive bool) {
•       var mu sync.Mutex
•       queue := []types.IngredientPair{{target}}
•       parent := -2
•
•       for len(queue) > 0 {
•           currentNode := queue
•           queue = nil
•
•           var wg sync.WaitGroup
•           for _, currentNode := range currentNode {
•               mu.Lock()
•               parent++
•               localParent := parent
•               mu.Unlock()
•               for _, element := range currentNode {
•                   if graph.IsLeaf(element) || element == "" {
•                       continue
•                   }
•
•                   recipes := graph.Graph[element]
•                   listRecipe := graph.FilterTier(element)
•                   mu.Lock()
•                   tempNRecipe := nRecipe
•                   mu.Unlock()
•                   if (tempNRecipe == 0){
•                       tempNRecipe = 1
•                   }
•                   limit := int(math.Min(float64(tempNRecipe),
•                     float64(len(listRecipe))))
•
•                   for i := 0; i < limit; i++ {
•                       rec := listRecipe[i]
•                       wg.Add(1)
•
•                       go func(rec types.IngredientPair, output string) {
•                           defer wg.Done()
•
•                           if graph.Graph[rec[0]].Tier >= recipes.Tier ||
•                             graph.Graph[rec[1]].Tier >= recipes.Tier {
•                               return
•                           }
•
•                           newCombo := types.Combo{
•                               ID:      -1,
•                               ParentId: localParent,
•                               Inputs:  []string{rec[0], rec[1]},
•                               Output:  output,
•                           }
•
•                           mu.Lock()
•                           newCombo.ID = len(*combos)
•                           *combos = append(*combos, newCombo)
•                           queue = append(queue, rec)
•                           nRecipe--
•                           mu.Unlock()
•
•                           if isLive {
•                               progressChan <- newCombo
•                               time.Sleep(1 * time.Second)
•                           }
•                       }
•                   }
•               }
•           }
•       }
•   }

```

```

        }
    }(rec, element)
}
mu.Lock()
nRecipe++
mu.Unlock()
}
wg.Wait()
}
}

```

- DFS

```

func GetRecipeDFS(graph *types.RecipeGraph, currentNode types.IngredientPair,
combos *[]types.Combo, nRecipe *int, progressChan chan types.Combo, isLive
bool) {
}

var mutex sync.Mutex
var wg sync.WaitGroup

dfsWithConcurrency(graph, currentNode, combos, nRecipe, &mutex, &wg,
progressChan, isLive)

wg.Wait()
close(progressChan)
}

func dfsWithConcurrency(graph *types.RecipeGraph, currentNode
types.IngredientPair, combos *[]types.Combo,
nRecipe *int, mu *sync.Mutex, wg *sync.WaitGroup, progressChan chan
types.Combo, isLive bool) {
parent := len(*combos) - 1

for _, element := range currentNode {
    if graph.IsLeaf(element) || element == ""{
        continue
    }

    mu.Lock()
    *nRecipe += 1

    mu.Unlock()

    pairs := graph.FilterTier(element)

    for _, pair := range pairs {
        mu.Lock()
        if *nRecipe <= 0 {
            mu.Unlock()
            continue
        }

        (*nRecipe)--
        mu.Unlock()
        wg.Add(1)
        go func(p types.IngredientPair) {
            defer wg.Done()
            mu.Lock()
        }
    }
}

```

```

•          comboID := len(*combos)
•          newCombo := types.Combo{
•              ID:           comboID,
•              ParentId:    parent,
•              Inputs:      []string{p[0], p[1]},
•              Output:      element,
•          }
•          *combos = append(*combos, newCombo)
•          mu.Unlock()
•          if(isLive){
•              progressChan <- newCombo
•
•              time.Sleep(1000 * time.Millisecond)
•          }
•          dfsWithConcurrency(graph, p, combos, nRecipe, mu, wg,
•              progressChan, isLive)
•              }(pair)
•          }
•      }
}

```

Berikut adalah beberapa fungsi dan prosedur yang terdapat pada Frontend dan berhubungan langsung dengan pengolahan data hasil dari Backend.

- App.jsx

```

•      const stableCombos = useMemo(() => results?.combos || [], [results]);
•
•      const basicElements = useMemo(() => {
•          const allOutputs = new Set(stableCombos.map((c) => c.output));
•          const allInputs = new Set(
•              stableCombos.flatMap((c) => (Array.isArray(c.inputs) ? c.inputs : []))
•          );
•          return [...allInputs].filter((input) => !allOutputs.has(input));
•      }, [stableCombos]);
•
•      const [liveUpdateEnabled, setLiveUpdateEnabled] = useState(false);
•
•      const { allRecipeTrees, totalRecipePaths, totalRecipesUnfiltered } =
•          useMemo(() => {
•              function cartesianProduct(arrays) {
•                  return arrays.reduce(
•                      (a, b) =>
•                          a.length === 0 || b.length === 0
•                          ? []
•                          : a.flatMap((d) =>
•                              b.map((e) => (Array.isArray(d) ? [...d, e] : [d, e]))
•                          ),
•                      [[]]
•                  );
•              }
•
•              const uniqueCombosMap = new Map();
•              stableCombos.forEach((combo) => {
•                  if (!combo || !Array.isArray(combo.inputs)) return;
•                  const key = combo.output + "|" + [...combo.inputs].sort().join("+");
•                  if (!uniqueCombosMap.has(key)) uniqueCombosMap.set(key, combo);
•              });
•              const uniqueCombos = Array.from(uniqueCombosMap.values());
•
•          }, [stableCombos]);

```

```

•     const uniqueRecipesByOutput = new Map();
•     uniqueCombos.forEach((combo) => {
•       if (!uniqueRecipesByOutput.has(combo.output)) {
•         uniqueRecipesByOutput.set(combo.output, []);
•       }
•       uniqueRecipesByOutput.get(combo.output).push(combo);
•     });
•
•     const BASIC_ELEMENTS = (() => {
•       const allOutputs = new Set(uniqueCombos.map((c) => c.output));
•       const allInputs = new Set(
•         uniqueCombos.flatMap((c) => (Array.isArray(c.inputs) ? c.inputs : []))
•       );
•       return [...allInputs].filter((input) => !allOutputs.has(input));
•     })();
•
•     function buildRecipeTreesFromUnique(currentElement) {
•       if (BASIC_ELEMENTS.includes(currentElement)) {
•         return [
•           { name: currentElement, attributes: { type: "Basic Element" } },
•         ];
•       }
•
•       const possibleRecipes = uniqueRecipesByOutput.get(currentElement) || [];
•       if (possibleRecipes.length === 0) return [];
•
•       const treesForCurrentElement = [];
•       possibleRecipes.forEach((recipe) => {
•         const childrenTreeOptions = recipe.inputs.map((input) =>
•           buildRecipeTreesFromUnique(input)
•         );
•         if (childrenTreeOptions.some((options) => options.length === 0))
•           return;
•         const allCombinedChildrenSets = cartesianProduct(childrenTreeOptions);
•         allCombinedChildrenSets.forEach((childrenCombination) => {
•           treesForCurrentElement.push({
•             name: recipe.output,
•             attributes: { type: "Combined" },
•             children: childrenCombination,
•           });
•         });
•       });
•
•       return treesForCurrentElement;
•     }
•
•     function getTreeSignature(node) {
•       if (!node) return "";
•       let signature = node.name;
•       if (node.children && node.children.length > 0) {
•         const sortedChildrenSignatures = node.children
•           .map(getTreeSignature)
•           .sort()
•           .join(",");
•         signature += `(${sortedChildrenSignatures})`;
•       }
•       return signature;
•     }
•
•     const targetElement = currentSearch?.targetElement;
•     if (!targetElement)

```

```

•         return {
•             allRecipeTrees: [],
•             totalRecipePaths: 0,
•             totalRecipesUnfiltered: 0,
•         };
•
•     if (BASIC_ELEMENTS.includes(targetElement)) {
•         return {
•             allRecipeTrees: [
•                 {
•                     name: targetElement,
•                     attributes: { type: "Basic Element" },
•                     children: [],
•                 },
•                 ],
•                 totalRecipePaths: 1,
•                 totalRecipesUnfiltered: 1,
•             };
•         }
•
•         const rawTrees = buildRecipeTreesFromUnique(targetElement);
•         const uniqueTrees = [];
•         const seen = new Set();
•         rawTrees.forEach((t) => {
•             const sig = getTreeSignature(t);
•             if (!seen.has(sig)) {
•                 seen.add(sig);
•                 uniqueTrees.push(t);
•             }
•         });
•
•         const count = uniqueTrees.length;
•         const max = currentSearch?.maxRecipes ?? count
•         return {
•             allRecipeTrees: uniqueTrees.slice(0, max),
•             totalRecipePaths: Math.min(count, max),
•             totalRecipesUnfiltered: count,
•         };
•     }, [stableCombos, currentSearch?.targetElement]);
•

```

- RecipeTree.jsx

```

• import React, { useEffect } from "react";
• import Tree from "react-d3-tree";
•
• const RecipeTree = ({ treeData, elementName }) => {
•     useEffect(() => {
•         console.log("📌 Debug treeData:", treeData);
•     }, [treeData]);
•
•     if (!treeData && elementName) {
•         return (
•             <div className="w-full h-[600px] bg-[#fef9c3] border border-yellow-700 rounded-xl shadow-inner flex flex-col items-center justify-center space-y-4">
•                 <div className="px-8 py-4 bg-amber-50 text-3xl font-bold text-yellow-900 border-4 border-yellow-600 rounded-full shadow-md">
•                     {elementName}
•                 </div>
•                 <p className="text-yellow-800 font-merriweather italic text-center">
•                     Ini adalah elemen dasar tanpa resep kombinasi.
•                 </p>
•             </div>
•         );
•     }
•
•     return (
•         <div className="w-full h-[600px] bg-[#fef9c3] border border-yellow-700 rounded-xl shadow-inner flex flex-col items-center justify-center space-y-4">
•             <div className="px-8 py-4 bg-amber-50 text-3xl font-bold text-yellow-900 border-4 border-yellow-600 rounded-full shadow-md">
•                 {elementName}
•             </div>
•             <p className="text-yellow-800 font-merriweather italic text-center">
•                 Ini adalah elemen dasar tanpa resep kombinasi.
•             </p>
•         </div>
•     );
• }

```

```

        </p>
    );
}

if (
    typeof treeData === "object" &&
    !Array.isArray(treeData) &&
    treeData?.attributes?.type === "Basic Element"
) {
    return (
        <div className="w-full h-[600px] bg-[#fef9c3] border border-yellow-700 rounded-xl shadow-inner flex flex-col items-center justify-center space-y-4">
            <div className="px-8 py-4 bg-amber-50 text-3xl font-bold text-yellow-900 border-4 border-yellow-600 rounded-full shadow-md">
                {treeData.name}
            </div>
            <p className="text-yellow-800 font-merriweather italic text-center">
                Menampilkan elemen dasar tanpa resep kombinasi.
            </p>
        </div>
    );
}

return (
    <div className="w-full h-[600px] bg-[#fef9c3] border border-yellow-700 rounded-xl shadow-inner">
        <Tree
            data={treeData}
            orientation="vertical"
            pathFunc="step"
            zoomable
            separation={{ siblings: 1.5, nonSiblings: 2 }}
        />
    </div>
);
};

export default RecipeTree;

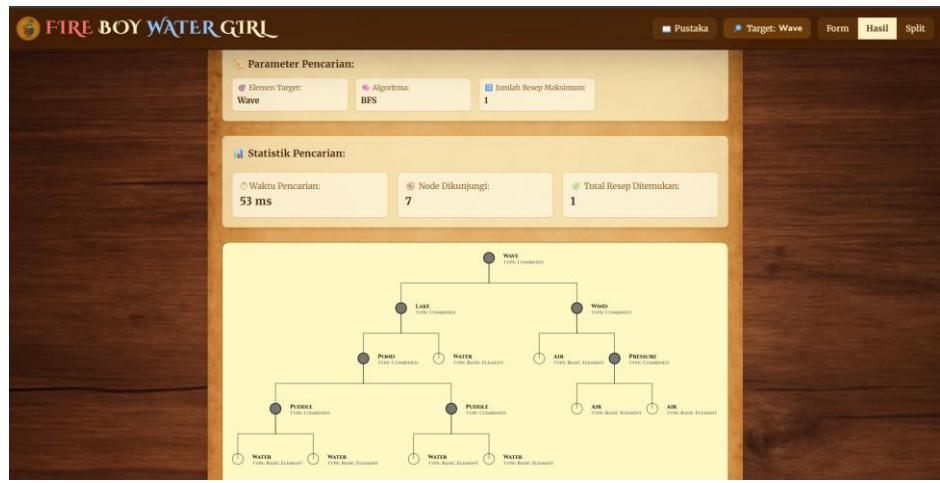
```

4.2 Tata Cara Penggunaan Program

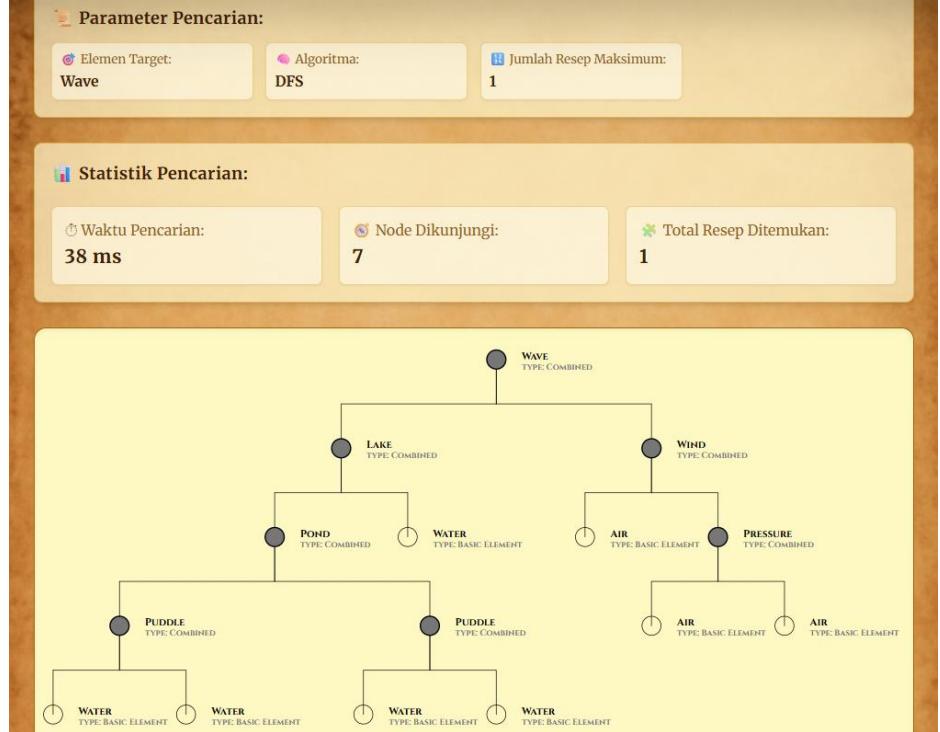
Web telah dideploy sehingga Anda bisa menjalankannya dengan mengakses link berikut Akses Web [Fireboy Watergirl](#).

4.3 Hasil Pengujian

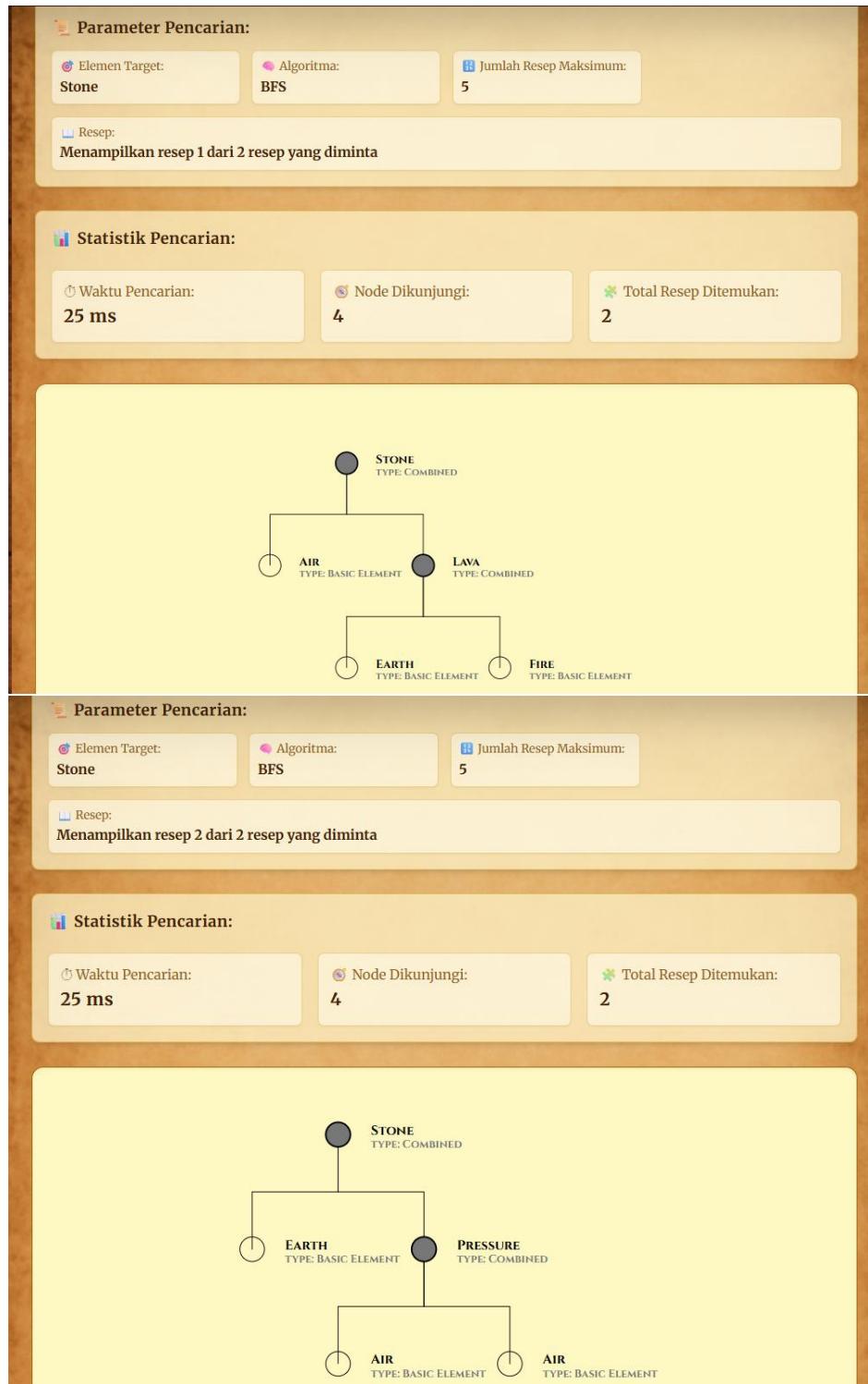
- Uji pencarian elemen Wave dengan menggunakan algoritma BFS dan jumlah resep satu



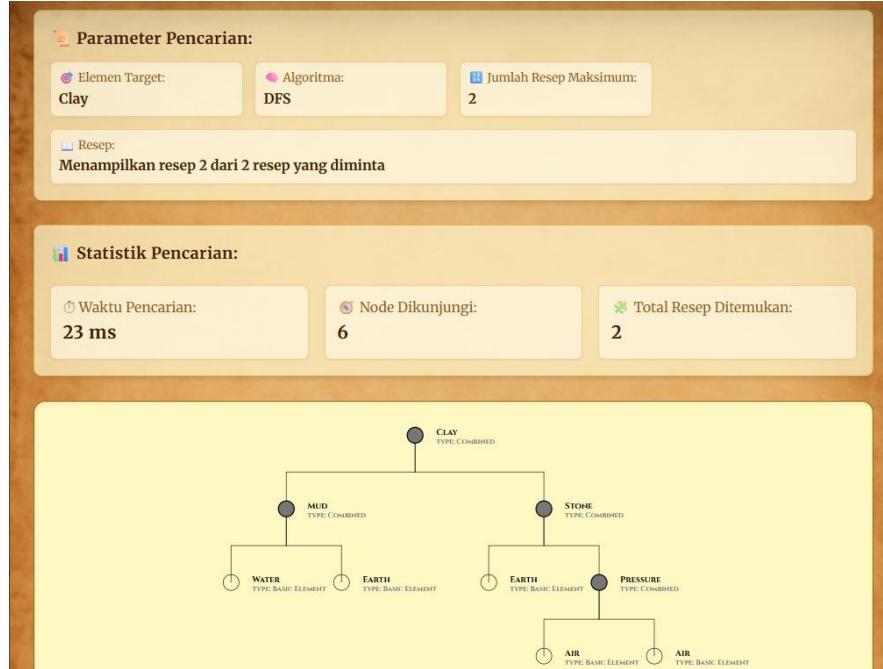
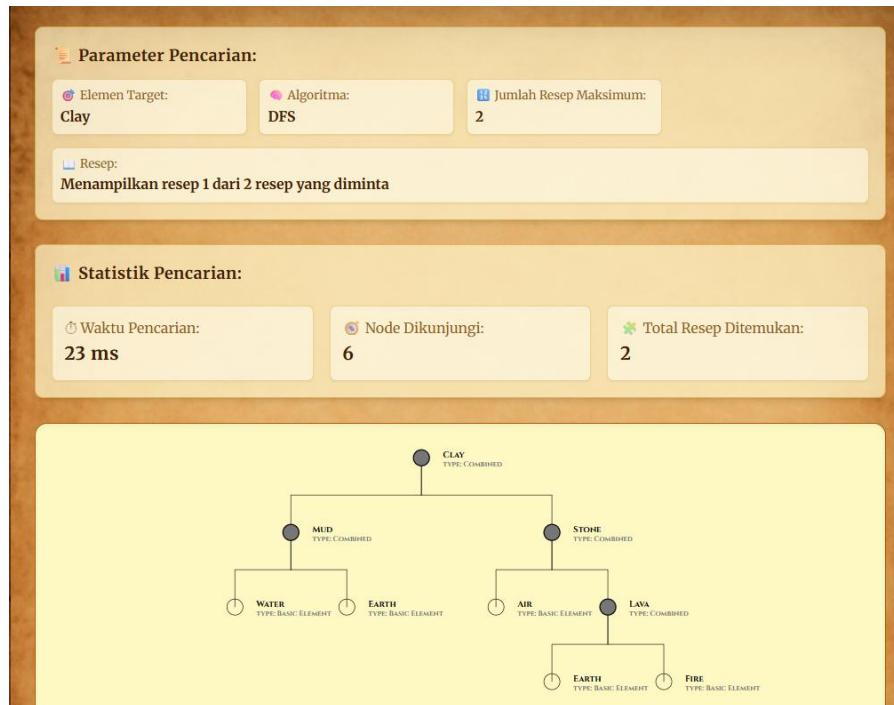
- Uji pencarian elemen Wave dengan menggunakan algoritma DFS dan jumlah resep satu.



- Uji pencarian elemen Stone menggunakan algoritma BFS dan jumlah resep dua.



- Uji pencarian elemen Clay menggunakan algoritma DFS dan jumlah resep dua.



- Uji pencarian

Parameter Pencarian:

- Elemen Target: Bird
- Algoritma: BFS
- Jumlah Resep Maksimum: 1

LIVE UPDATE VISUALISASI
PROSES PENCARIAN SEDANG BERLANGSUNG DAN AKAN DIPERBARUI SECARA BERTAHAP.

Visualisasi Progresif:

```

BIRD ← ANIMAL + AIR      BIRD ← ANIMAL + SKY      SKY ← SUN + MOON      ANIMAL ← LIFE + BEACH
ANIMAL ← LIFE + LAND     ANIMAL ← LIFE + MOUNTAIN    ANIMAL ← LIFE + MOUNTAIN RANGE
ANIMAL ← LIFE + DESERT    ANIMAL ← LIFE + BEACH     ANIMAL ← LIFE + MOUNTAIN RANGE
SKY ← SUN + ATMOSPHERE   ANIMAL ← LIFE + DESERT    ANIMAL ← LIFE + LAND
ANIMAL ← LIFE + MOUNTAIN  LIFE ← PRIMORDIAL SOUP + ENERGY  LIFE ← PRIMORDIAL SOUP + ENERGY
DESERT ← SAND + SAND      DESERT ← SAND + LAND     LIFE ← PRIMORDIAL SOUP + VOLCANO
LIFE ← PRIMORDIAL SOUP + ENERGY  BEACH ← SAND + LAKE  BEACH ← SAND + WATER
LIFE ← PRIMORDIAL SOUP + VOLCANO  LIFE ← PRIMORDIAL SOUP + ENERGY
MOUNTAIN RANGE ← MOUNTAIN + MOUNTAIN  MOUNTAIN RANGE ← MOUNTAIN + CONTINENT

```

Parameter Pencarian:

- Elemen Target: Bird
- Algoritma: BFS
- Jumlah Resep Maksimum: 1

Statistik Pencarian:

- Waktu Pencarian: null ms
- Node Dikunjungi: 133
- Total Resep Ditemukan: 57200

*Note : waktu pada visualisasi live update bernilai null karena pada fitur tersebut tidak fokus dalam menghitung atau melihat kecepatan proses algoritma, melainkan fokus pada proses dekomposisi.

4.4 Analisis Hasil Pengujian

Dari hasil pengujian, didapatkan hasil dari algoritma BFS maupun DFS untuk single recipe atau multiple recipe selesai dalam hitungan ms. Hal tersebut dapat terjadi karena pada backend menggunakan teknik multithreading sehingga mempercepat proses pencarian.

5 KESIMPULAN

5.1 Kesimpulan

Algoritma BFS dan DFS yang diimplementasikan dalam proyek FireBoyWaterGirl terbukti efektif untuk mencari resep elemen dalam permainan Little Alchemy 2. Pendekatan pencarian berbasis graf ini memungkinkan pengguna menemukan jalur kombinasi dari elemen dasar untuk mencapai elemen target yang diinginkan. Dari analisis kode yang ada, implementasi BFS menelusuri graf secara melebar level per level, sementara DFS mengikuti satu jalur hingga akhir, keduanya memberikan solusi yang valid walaupun dengan karakteristik berbeda. Sistem visualisasi dalam bentuk tree yang dibangun memudahkan pengguna memahami urutan kombinasi elemen. Penggunaan multithreading pada mode pencarian multiple recipe meningkatkan performa aplikasi terutama saat mencari banyak resep alternatif. Fitur live update visualisasi memberikan pengalaman interaktif yang membantu pengguna memahami proses pencarian yang sedang berlangsung.

5.2 Saran

1. Pengembangan visualisasi yang lebih informatif dengan menambahkan informasi tier pada setiap elemen untuk memahami tingkat kesulitannya.
2. Implementasi caching hasil pencarian untuk mempercepat pencarian yang berulang.
3. Optimasi penggunaan memori pada algoritma BFS dan DFS, terutama saat menangani dataset elemen yang besar.
4. Menambahkan tutorial interaktif tentang cara kerja algoritma BFS/DFS dalam konteks aplikasi ini.
5. Implementasi fitur ekspor hasil pencarian dalam format yang dapat dibagikan.

5.3 Komentar

Tugas besar ini memberikan banyak pengalaman pelajaran bagi kami. Dengan adanya tugas besar ini, kami jadi mengetahui bagaimana memrogram dengan bahasa Go dan *framework* pemrograman React dan juga menggunakan API. Kami juga menjadi lebih paham dan tahu cara menerapkan konsep algoritma BFS dan DFS yang telah kami pelajari di kelas.

5.4 Refleksi

Kami menyadari beberapa hal penting selama pengerjaan tugas ini:

1. Pentingnya perencanaan awal dalam mendefinisikan struktur data yang tepat untuk merepresentasikan elemen dan resep.
2. Bagaimana mengoptimalkan algoritma BFS dan DFS untuk kasus spesifik pencarian resep dengan batasan tier.
3. Tantangan dalam menampilkan visualisasi hasil pencarian secara efektif dan informatif.
4. Perlunya koordinasi yang baik antara pengembangan frontend dan backend, terutama dalam mendefinisikan format data yang dipertukarkan.

5. Nilai lebih dari fitur live update yang memungkinkan pengguna memahami proses pencarian secara bertahap.
6. Pentingnya testing secara menyeluruh dengan berbagai kasus uji untuk memastikan kebenaran hasil pencarian.

6 LAMPIRAN

6.1 Tautan Repository Github

- Repository Frontend
https://github.com/Rusmn/Tubes2_FE_FireBoyWaterGirl
- Repository Backend
https://github.com/AryoBama/Tubes2_BE_FireBoyWaterGirl

6.2 Tautan Video

https://youtu.be/K5qcX8M_27Y

7 DAFTAR PUSTAKA

<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>

<https://informatika.stei.itb.ac.id/~rinaldi.munir>

<https://www.trivusi.web.id/2022/05/apa-itu-algoritma-depth-first-search.html>

No	Poin	Ya	Tidak
1	Aplikasi dapat dijalankan.	✓	
2	Aplikasi dapat memperoleh data <i>recipe</i> melalui scraping.	✓	
3	Algoritma <i>Depth First Search</i> dan <i>Breadth First Search</i> dapat menemukan <i>recipe</i> elemen dengan benar.	✓	
4	Aplikasi dapat menampilkan visualisasi <i>recipe</i> 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.	✓	