

LAPORAN TUGAS BESAR 2
IF2211-24 STRATEGI ALGORITMA



Kelompok 21 Deadliner Tobat

Muhammad Raihan Nazhim Oktana 13523021

Mayla Yaffa Ludmilla 13523050

Anella Utari Gunadi 13523078

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

DAFTAR ISI

DAFTAR ISI.....	1
BAB 1 Deskripsi Tugas.....	2
BAB 2 Landasan Teori.....	3
2.1 Penjelajahan Graf.....	3
2.2 Algoritma Breadth First Search (BFS).....	3
2.3 Algoritma Depth First Search (DFS).....	4
2.4 Algoritma Bidirectional.....	4
2.5 Gambaran Umum Aplikasi Web.....	5
BAB 3 Analisis Pemecahan Masalah.....	5
3.1 Langkah-langkah Pemecahan Masalah.....	5
3.1.1 Identifikasi Permasalahan.....	5
3.1.2 Pemilihan Struktur Data.....	5
3.1.3 Desain Algoritma.....	6
3.1.4 Optimasi Program.....	6
3.2 Proses Pemetaan Masalah Menjadi Elemen Algoritma BFS & DFS.....	6
3.3 Fitur Fungsional dan Arsitektur Aplikasi Web.....	7
3.3.1 Fitur Fungsional Aplikasi Web.....	7
3.3.2 Arsitektur Aplikasi Web.....	7
3.4 Ilustrasi Kasus.....	8
BAB 4 Implementasi dan Pengujian.....	9
4.1 Spesifikasi Teknis dan Struktur Program.....	9
4.1.1 Struktur Data.....	9
4.1.2 Fungsi Dan Prosedur Utama.....	10
4.1.3 Fungsi Dan Prosedur Pendukung.....	22
4.2 Antarmuka dan Penggunaan Aplikasi.....	27
4.2.1 Antarmuka Aplikasi.....	27
4.2.2 Fitur-fitur yang Disediakan Aplikasi.....	27
4.2.3 Tata Cara Penggunaan Aplikasi.....	30
4.3 Hasil Pengujian.....	31
4.4 Analisis Hasil Pengujian.....	36
BAB 5 Kesimpulan, Saran, dan Refleksi.....	37
5.1 Kesimpulan.....	37
5.2 Saran.....	37
5.3 Refleksi.....	38
Lampiran.....	38
Daftar Pustaka.....	39

BAB 1 Deskripsi Tugas

Dalam tugas besar 2 ini, kami diminta untuk membuat suatu permainan berbasis aplikasi *website* yang bernama Little Alchemy 2 dengan memanfaatkan algoritma *Breadth First Search* dan *Depth First Search* (serta *Bidirectional* untuk spesifikasi bonus). 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. Elemen dasar pada Little Alchemy 2

2. Elemen turunan

Terdapat 720 elemen turunan yang dibagi menjadi beberapa *tier* tergantung tingkat kesulitan dan banyak langkah yang harus dilakukan. Setiap elemen turunan memiliki *recipe* yang terdiri atas elemen lainnya atau elemen itu sendiri.

3. *Combine Mechanism*

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

BAB 2 Landasan Teori

2.1 Penjelajahan Graf

Dalam kasus ini, graf merupakan representasi persoalan. Penjelajahan graf artinya melakukan pencarian solusi persoalan yang direpresentasikan dalam bentuk graf. Ada dua algoritma pencarian solusi berbasis graf yang ada, yaitu tanpa informasi (uninformed/blind search) dan dengan informasi (informed search). Pada pencarian tanpa informasi, tidak ada informasi tambahan yang disediakan. Contohnya adalah pada algoritma *Breadth First Search*, *Depth First Search*, *Depth Limited Search*, *Iterative Deepening Search*, dan *Uniform Cost Search*. Pada pencarian dengan informasi, pencarian dilakukan dengan berbasis suatu heuristik tertentu sehingga bisa mengetahui *non-goal state* yang “lebih menjanjikan” dibandingkan yang lain. Contohnya adalah pada algoritma *Best First Search* dan A*.

Pada penjelajahan graf, ada dua tipe representasi graf yaitu graf statis dan graf dinamis. Graf statis adalah graf yang sudah terbentuk sebelum proses pencarian dilakukan dan direpresentasikan sebagai struktur data. Graf dinamis adalah graf yang terbentuk saat proses pencarian dilakukan dan tidak tersedia sebelum pencarian melainkan dibangun selama pencarian solusi.

2.2 Algoritma *Breadth First Search* (BFS)

Breadth-First Search (BFS) adalah algoritma *uninformed search* yang menelusuri simpul-simpul graf secara *layer-by-layer*: mulai dari simpul sumber, kemudian seluruh tetangganya, lalu tetangga dari setiap tetangga, dan seterusnya.

Algoritma BFS:

1. Kunjungi simpul v, tandai dikunjungi.
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.
4. Ulangi sampai tidak ada simpul yang belum dikunjungi atau simpul tujuan sudah dikunjungi.

2.3 Algoritma *Depth First Search* (DFS)

Depth-First Search (DFS) menelusuri graf secara “sedalam mungkin” di setiap cabang sebelum backtrack ke simpul terdahulu. Ketika kita mengunjungi sebuah simpul tetangga, kita menuntaskan terlebih dahulu penelusuran seluruh simpul lain yang dapat dicapai melalui simpul tetangga tersebut.

Algoritma DFS:

1. Kunjungi simpul v , tandai sebagai sudah dikunjungi.
2. Pilih satu simpul tetangga dari v yang belum dikunjungi.
3. Lanjutkan penelusuran ke simpul tersebut secara rekursif
4. Jika semua tetangga sudah dikunjungi, kembali (backtrack) ke simpul terdekat yang belum ditelusuri.
5. Ulangi langkah 2–4 sampai semua simpul yang dapat dicapai telah dikunjungi atau simpul tujuan ditemukan.

2.4 Algoritma *Bidirectional*

Bidirectional Search (Pencarian Dua Arah) adalah teknik pencarian pada graf yang dilakukan secara simultan dari dua arah: dari simpul awal (*start node*) dan dari simpul tujuan (*goal node*), hingga kedua pencarian bertemu di tengah. Pendekatan ini dapat mengurangi ruang pencarian secara signifikan, terutama pada graf besar, karena masing-masing pencarian hanya perlu menjelajahi sebagian dari keseluruhan ruang.

Dalam graf tak berbobot, algoritma ini sangat efisien dibandingkan pencarian satu arah seperti BFS atau DFS karena kompleksitas algoritma Bidirectional mendekati $O(b^{d/2})$ dibanding $O(b^d)$ pada BFS dan DFS, dengan b adalah branching factor dan d adalah depth.

Algoritma Bidirectional:

1. Inisialisasi dua antrian: satu dari simpul awal dan satu dari simpul tujuan.
2. Jalankan pencarian BFS dari kedua arah secara bergantian.
3. Tandai setiap simpul yang dikunjungi dalam dua himpunan terpisah (dari awal dan dari tujuan).
4. Setiap kali sebuah simpul baru dikunjungi, periksa apakah simpul tersebut sudah pernah dikunjungi dari arah lain.
5. Jika ditemukan simpul yang dikunjungi dari kedua arah, hentikan pencarian dan gabungkan jalur dari dua arah tersebut.
6. Jika tidak ada jalur yang ditemukan dan semua kemungkinan sudah habis, maka simpul tujuan tidak dapat dicapai dari simpul awal.

2.5 Gambaran Umum Aplikasi Web

Aplikasi web yang dibangun merupakan alat bantu visual untuk menelusuri *resep* pembentukan elemen dalam permainan Little Alchemy 2. Aplikasi ini memungkinkan pengguna untuk mencari cara membuat suatu elemen dari elemen-elemen dasar lainnya menggunakan pendekatan algoritma pencarian graf.

Fitur utama dari aplikasi ini meliputi:

- Pencarian elemen dengan berbagai algoritma penelusuran graf: Breadth-First Search (BFS), Depth-First Search (DFS), dan Bidirectional Search.
- Visualisasi *recipe tree* dalam bentuk pohon interaktif untuk membantu pengguna memahami jalur pembentukan elemen.
- Opsi pengaturan jumlah *recipe* yang ditampilkan agar hasil pencarian tidak terlalu kompleks.
- Tampilan antarmuka yang interaktif dan responsif, dibangun dengan Next.js dan React.js, serta komunikasi data dengan backend yang dibangun menggunakan Golang.

Aplikasi ini ditujukan untuk memberikan pengalaman eksploratif dan edukatif bagi pengguna, terutama dalam memahami penerapan algoritma penelusuran pada struktur graf yang kompleks dan bercabang.

BAB 3 Analisis Pemecahan Masalah

3.1 Langkah-langkah Pemecahan Masalah

Pada program ini, dilakukan analisis pemecahan masalah menjadi beberapa bagian, yaitu identifikasi permasalahan, pemilihan struktur data, desain algoritma, dan optimasi. Secara detail, hasil analisis yang dilakukan yaitu sebagai berikut:

3.1.1 Identifikasi Permasalahan

1. Program ingin mencari semua kemungkinan *recipe* untuk menghasilkan elemen target dari elemen base berdasarkan game Little Alchemy 2.
2. Tantangan yang menyusahkan adalah banyaknya elemen, banyaknya kemungkinan kombinasi, dan harus mencegah terhitungnya duplikasi atau terjebak dalam loop.

3.1.2 Pemilihan Struktur Data

1. Element : Membuat data statis setiap elemen, berisi Name, Tier, dan Parents.
2. Gallery : Menggunakan map, memetakan name menuju pointer terhadap Element.
3. RecipeNode : Membentuk node tree, dengan Name dan Parents []*RecipeNode.

4. PartialTree : Digunakan pada BFS, mencatat simpul dan daftar leaf yang perlu di-expand.
5. AlgorithmResult : Digunakan pada semua, menyimpan bentuk return di pencarian utama.

3.1.3 Desain Algoritma

1. BFS : Melakukan penjelajahan recipe secara melebar terlebih dulu, menggunakan *queue* untuk looping elemen dan *expandable leaf* untuk melebarkan pencarian dan *signature tree* untuk mencegah duplikat.
2. DFS : Melakukan penjelajahan recipe secara mendalam terlebih dulu sampai tuntas, menggunakan *stack* untuk rekursif dan konsep *memory memoization* untuk mempercepat pencarian dan mencegah duplikat.
3. BDR : Melakukan penjelajahan recipe secara 2 arah dari top dan bottom dengan konsep *meet-in-the-middle*, membangun tree dari root target ke daerah mid-tier dan dari base element ke mid-tier, lalu menggabungkan kedua tree tersebut.

3.1.4 Optimasi Program

1. Membatasi eksplorasi hanya pada parent dengan tier lebih rendah dari elemen saat ini.
2. Melakukan pencegahan duplikasi dengan *signature* dan *memory memoization*.
3. Melakukan multithreading untuk proses paralel menggunakan goroutine yang lebih cepat.

3.2 Proses Pemetaan Masalah Menjadi Elemen Algoritma BFS & DFS

Dalam menyelesaikan permasalahan ini, dilakukan pemetaan masalah menjadi elemen penting dalam algoritma BFS & DFS yang diimplementasikan, sehingga secara garis besar dapat menggambarkan bagaimana masalah program didekomposisi dan dirancang sebagai berikut:

Peta Masalah	BFS	DFS
Node Exploration	Menggunakan looping utama for ($\text{len}(\text{queue}) > 0$).	Menggunakan fungsi rekursif <i>EnumerateDFS</i> .
Penyimpanan State	Menggunakan <i>parent_map</i> untuk <i>caching parent_pairs</i> .	Menggunakan <i>stack</i> ($\text{map}[\text{string}]\text{bool}$) Untuk melakukan <i>cycle-detection</i> .
Memoization	Menggunakan <i>signature_tree</i> ($\text{map}[\text{string}]\text{struct} \{ \}$) untuk mencegah output yang duplikat.	Menggunakan <i>memory</i> ($\text{map}[\text{string}]\text{[*RecipeNode]}$) untuk mencegah output yang duplikat.

Runut Balik (backtracking)	Melakukan <i>pop_front</i> dari <i>queue</i> setelah proses dilakukan.	Melakukan unmark pada <i>stack</i> dengan <i>stack[name] = false</i> setelah rekursi.
Penggabungan Hasil	Kumpulan <i>res</i> []* <i>RecipeNode</i> diisi saat (<i>len(new_leaf)</i> == 0) sebelum melanjutkan enqueue berikutnya.	Kumpulan <i>res</i> []* <i>RecipeNode</i> yang dikembalikan oleh fungsi rekursif setelah proses selesai.

3.3 Fitur Fungsional dan Arsitektur Aplikasi Web

3.3.1 Fitur Fungsional Aplikasi Web

Aplikasi web ini dirancang untuk dapat mencari resep pembuatan suatu elemen berdasarkan elemen-elemen penyusunnya. Proses pencarian dilakukan dengan menerapkan algoritma penelusuran graf, yaitu *Breadth First Search* (BFS) dan *Depth First Search* (DFS), dan *Bidirectional*. Aplikasi ini menyediakan fitur-fitur berikut:

- Pencarian Elemen Berdasarkan Nama

Pengguna dapat mengetikkan nama elemen yang akan dicari, kemudian sistem akan mencari jalur pembentukannya (*recipe*).

- Pemilihan Algoritma Penelusuran

Pengguna dapat memilih salah satu dari tiga algoritma, yaitu BFS, DFS, atau Bidirectional.

- Pengaturan Jumlah Resep Maksimal

Aplikasi memungkinkan pengguna menentukan jumlah maksimal resep yang ingin ditampilkan untuk mempercepat pencarian dan menghindari overload visualisasi.

- Visualisasi Tree Resep

Hasil pencarian ditampilkan dalam bentuk pohon (*tree*) yang menunjukkan urutan kombinasi elemen untuk membentuk elemen target.

- Feedback Waktu Eksekusi dan Jumlah Node

Aplikasi menampilkan waktu proses pencarian dan jumlah node yang dikunjungi untuk memberi insight tentang efisiensi tiap algoritma.

3.3.2 Arsitektur Aplikasi Web

- Frontend

Frontend dari aplikasi ini menggunakan framework **Next.js** dan bahasa **JavaScript**, serta diperkantik dengan **Tailwind CSS**. Frontend bertanggung jawab atas:

- Tampilan antarmuka pengguna (UI)
- Form input elemen dan pengaturan pencarian
- Pemilihan algoritma
- Visualisasi tree hasil pencarian menggunakan pustaka react-d3-tree

- Backend

Backend dari aplikasi ini menggunakan bahasa Golang untuk mencari resep elemen menggunakan metode BFS, DFS, dan bidirectional. API menggunakan library Gin. Backend bertugas untuk:

- Menyimpan dan memuat data elemen dan resep dari file JSON
- Melakukan pencarian menggunakan algoritma BFS, DFS, dan Bidirectional
- Mengembalikan hasil pencarian dalam bentuk struktur tree kepada frontend

- Data Source

Data elemen Little Alchemy 2 diambil menggunakan teknik *web-scraping* dari laman Wiki Alchemy 2 yang diubah menjadi bentuk JSON. Dalam aplikasi ini, elemen-elemen yang merupakan bagian dari paket “Myths and Monsters” tidak diperhitungkan sebagai resep. *Web-scraping* menggunakan library goquery.

3.4 Ilustrasi Kasus

- Pengguna ingin mengetahui bagaimana cara membentuk elemen Galaxy, dengan detail:
 - Elemen yang dicari: Galaxy
 - Algoritma yang dipilih: Breadth First Search (BFS)
 - Jumlah maksimal recipe yang ingin dilihat: 2
 - Mode pencarian: Multiple Recipes
- Pengguna ingin mengetahui bagaimana cara membentuk elemen Rainbow, dengan detail:
 - Elemen yang dicari: Rainbow
 - Algoritma yang dipilih: Depth First Search (DFS)
 - Mode pencarian: Single Recipe
- Pengguna ingin mengetahui bagaimana cara membentuk elemen Rainbow, dengan detail:
 - Elemen yang dicari: Rainbow
 - Algoritma yang dipilih: Bidirectional
 - Mode pencarian: Single Recipe

BAB 4 Implementasi dan Pengujian

4.1 Spesifikasi Teknis dan Struktur Program

4.1.1 Struktur Data

```
type Element struct {
    Name    string;
    Tier    int;
    Parents [][]string;
}

type Gallery struct {
    GalleryName map[string]*Element;
}

type RecipeNode struct {
    Name    string      `json:"name"`;
    Parents []*RecipeNode `json:"children,omitempty"`;
}

type PartialTree struct {
    Tree  *RecipeNode;
    Leaf  []*RecipeNode;
}

type AlgorithmResult struct {
    Trees      []*RecipeNode;
    VisitedCount int;
}
```

Berdasarkan potongan kode diatas, kami membuat 5 type struct untuk memecah masalah yang kompleks ini menjadi lebih sederhana. Pada type struct Element, disimpan data Name, Tier, dan kumpulan semua kemungkinan 2 Parents-nya secara statis. Lalu, pada type struct Gallery, disimpan data GalleryName yang menggunakan map string menuju pointer terhadap Element key-nya. Lalu, pada type struct RecipeNode, disimpan data Name dan Parents dalam bentuk node untuk digunakan dalam pencarian yang dinamis. Lalu, pada type struct PartialTree, disimpan data Tree sebagai node yang utama dan sudah dieksekusi dan data Leaf sebagai kumpulan node lain yang perlu diekspansi dan dieksekusi dalam Algoritma BFS.

Serta, pada type struct yang terakhir yaitu AlgorithmResult, disimpan data Trees sebagai kumpulan Tree dari solusi yang mungkin, serta VisitedCount sebagai total node yang dikunjungi selama proses pencarian dilakukan.

4.1.2 Fungsi Dan Prosedur Utama

1. LoadRecipeGallery

```
func LoadRecipeGallery(path string) (*Gallery , error) {
    file , err := os.Open(path);
    if (err != nil) {
        return nil , err;
    } else {
        defer file.Close();
        raw := map[string]any{};
        if err := json.NewDecoder(file).Decode(&raw) ; err != nil {
            return nil , fmt.Errorf("decode JSON : %w" , err);
        } else {
            gallery := &Gallery{GalleryName : map[string]*Element{}};
            check := false;
            for key := range raw {
                if (strings.HasPrefix(strings.ToLower(key) , "tier")) {
                    check = true;
                    break;
                }
            }
            if (check) {
                for key , value := range raw {
                    if (strings.HasPrefix(strings.ToLower(key) ,
"tier")) {
                        str :=
strings.TrimSpace(strings.TrimPrefix(strings.ToLower(key) , "tier"));
                        num , _ := strconv.Atoi(str);
                        inner , check := value.(map[string]any);
                        if (check) {
                            for name , arr := range inner {
                                var parents [][]string;
                                if (arr != nil) {
                                    parents = Transform(arr.([]any));
                                }
                                gallery.GalleryName[name] = &Element{
                                    Name      : name,
                                    Tier      : num,
```

Berdasarkan potongan kode diatas, Fungsi ini membuka file JSON resep di path, lalu men-decode-nya menjadi raw map[string]any. Kemudian, dilakukan pemeriksaan pada file sudah dikelompokkan per-tier atau belum, lalu membangun gallery.GalleryName dengan memasukkan setiap elemen yang ada sebagai &Element{Name, Tier, Parents}

menggunakan fungsi bantuan Transform. Setelah itu, di-set base element (Water, Earth, Fire, Time, Air) ada dengan nilai tier 0, lalu memanggil CalculateTier secara rekursif untuk menghitung tier elemen lain yang belum terdefinisi, sehingga dihasilkan peta lengkap Element, Tier, dan daftar Parents yang siap dipakai di algoritma pencarian.

2. BFS (Breadth First Search)

```
func BFS(gallery *Gallery , target string , max_recipe int)
AlgorithmResult {
    if max_recipe == 0 {
        max_recipe = int(^uint(0) >> 1)
    }
    touch();
    if element , check := gallery.GalleryName[target] ; (check &&
element.Tier == 0) {
        node := &RecipeNode{Name : target};
        return AlgorithmResult{Trees : []*RecipeNode{node} ,
VisitedCount : int(atomic.LoadInt64(&counter))};
    } else {
        parent_map := make(map[string][][2]string);
        GetParentPairs := func(name string) [][]2]string {
            if parent_pairs , check := parent_map[name] ; check {
                return parent_pairs;
            } else {
                var all_parent_pairs [][]2]string;
                if element , check := gallery.GalleryName[name] ; check
{
                    for _ , parent := range element.Parents {
                        all_parent_pairs = append(all_parent_pairs ,
[2]string{parent[0] , parent[1]});
                    }
                }
                parent_map[name] = all_parent_pairs;
                return all_parent_pairs;
            }
        }
        var (
            res    []*RecipeNode;
            next   []PartialTree;
            queue  []PartialTree;
            mutex_res sync.Mutex;
            mutex_next sync.Mutex;
        )
        for _ , parent := range element.Parents {
            all_parent_pairs = append(all_parent_pairs ,
[2]string{parent[0] , parent[1]});
        }
        parent_map[name] = all_parent_pairs;
        return all_parent_pairs;
    }
}
```

```

        signature_tree = make(map[string]struct{}));
    )
    queue = append(queue , PartialTree{Tree : &RecipeNode{Name : target} , Leaf : []*RecipeNode{{Name : target}}});
    for (len(queue) > 0 && len(res) < max_recipe) {
        var wg sync.WaitGroup;
        next = next[:0];
        for _ , cur := range queue {
            exp := cur.Leaf[0];
            element := gallery.GalleryName[exp.Name];
            for _ , parent := range GetParentPairs(exp.Name) {
                touch();
                l := parent[0];
                r := parent[1];
                if (GetTier(gallery , l) < element.Tier &&
GetTier(gallery , r) < element.Tier) {
                    wg.Add(1);
                    go func(l string , r string , cur PartialTree)
{
                        defer wg.Done()
                        new_root , _ := CloneTreeMap(cur.Tree);
                        ptr := FindPointer(new_root ,
cur.Leaf[0].Name)
                        if (ptr == nil) {
                            return;
                        } else {
                            pl := &RecipeNode{Name : l};
                            pr := &RecipeNode{Name : r};
                            ptr.Parents = []*RecipeNode{pl , pr};
                            var new_leaf []*RecipeNode;
                            for _ , Leaf := range cur.Leaf[1:] {
                                if np := FindPointer(new_root ,
Leaf.Name) ; np != nil {
                                    new_leaf = append(new_leaf , np);
                                }
                            }
                            if
(IsExpandable(gallery.GalleryName[l])) {
                                new_leaf = append(new_leaf , pl);
                            }
                            if
(IsExpandable(gallery.GalleryName[r])) {

```

Berdasarkan potongan kode diatas, Algoritma BFS memulai dari satu `PartialTree{Tree: root, Leaf : [root]}` dan memperluas level kedalaman menggunakan queue. Pada setiap node `leaf exp` (pertama), dilakukan pengambilan semua pasangan `recipe (l, r)` dari `GetParentPairs`, yang cache di `parent_map`, lalu menjalankan goroutine dengan meng-clone tree saat ini menggunakan `CloneTreeMap`, mencari pointer leaf dengan `FindPointer`, lalu menyematkan dua child baru. Apabila leaf-leaf baru tidak expandable,

hasilnya diberikan signature dengan SignatureTree untuk menghindari duplikat yang disimpan di signature_tree dan hasilnya disimpan di res. Jika masih ada leaf, antrian berikutnya next diisi, dan setelah semua goroutine selesai, queue di-swap dengan next. Setiap kali melewati sebuah exp, fungsi touch() dipanggil untuk menambah counter atomik, yang kemudian dikembalikan sebagai VisitedCount. Mutex dilakukan untuk menjamin hanya ada satu goroutine saja yang boleh memegang kunci dalam satu waktu.

3. DFS (Depth First Search)

```
func DFS(gallery *Gallery , target string , max_recipe int)
AlgorithmResult {
    if (max_recipe == 0) {
        max_recipe = int(^uint(0) >> 1);
    }
    element := gallery.GalleryName[target];
    if (element == nil || element.Tier == 0 || len(element.Parents) == 0) {
        touch();
        node := &RecipeNode{Name : target};
        return AlgorithmResult{Trees : []*RecipeNode{node} ,
VisitedCount : int(atomic.LoadInt64(&counter))};
    } else {
        var (
            res    []*RecipeNode;
            mutex sync.Mutex;
            memory sync.Map;
        )
        var EnumerateDFS func(string , map[string]bool) []*RecipeNode;
        EnumerateDFS = func(name string , stack map[string]bool)
[*RecipeNode] {
            touch();
            if (stack[name]) {
                return []*RecipeNode{{Name : name}};
            } else if value , check := memory.Load(name) ; check {
                return CloneSlice(value.([]*RecipeNode));
            } else {
                element := gallery.GalleryName[name];
                if (element == nil || element.Tier == 0 || len(element.Parents) == 0) {
                    base := []*RecipeNode{{Name : name}};
                    memory.Store(name , base);
                    return base;
                } else {
                    for _, parent := range element.Parents {
                        stack[parent] = true;
                    }
                    go func() {
                        defer func() {
                            stack[element.Name] = false;
                        }()
                        res = append(res, &RecipeNode{Name : element.Name});
                        if (parent != nil) {
                            EnumerateDFS(parent, stack);
                        }
                    }()
                }
            }
        }
    }
}
```

```

    } else {
        stack[name] = true;
        var local_res []*RecipeNode;
        for _, parent := range element.Parents {
            l := parent[0];
            r := parent[1];
            if (GetTier(gallery , l) < element.Tier &&
GetTier(gallery , r) < element.Tier) {
                pl := EnumerateDFS(l , stack);
                pr := EnumerateDFS(r , stack);
                for _, ll := range pl {
                    for _, rr := range pr {
                        touch();
                        local_res = append(local_res ,
&RecipeNode{Name: name , Parents: []*RecipeNode{ll , rr}}));
                        if (len(local_res) >= max_recipe) {
                            memory.Store(name , local_res);
                            stack[name] = false;
                            return CloneSlice(local_res);
                        }
                    }
                }
            }
            memory.Store(name , local_res);
            stack[name] = false;
            return CloneSlice(local_res);
        }
    }
    var wg sync.WaitGroup;
    for _, parent := range element.Parents {
        l := parent[0];
        r := parent[1];
        if (GetTier(gallery , l) < element.Tier && GetTier(gallery ,
, r) < element.Tier) {
            wg.Add(1);
            go func(l string , r string) {
                defer wg.Done();
                local_stack := map[string]bool{};
                pl := EnumerateDFS(l , local_stack);
                pr := EnumerateDFS(r , local_stack);
                var local_res []*RecipeNode;

```

```

        for _, ll := range pl {
            for _, rr := range pr {
                touch();
                local_res = append(local_res ,
&RecipeNode{Name : target , Parents : []*RecipeNode{ll , rr}}));
                if (len(local_res) >= max_recipe) {
                    break;
                }
            }
            if (len(local_res) >= max_recipe) {
                break;
            }
        }
        mutex.Lock();
        res = append(res , local_res...);
        mutex.Unlock();
    }(l , r);
}
wg.Wait();
if (len(res) > max_recipe) {
    res = res[:max_recipe];
}
return AlgorithmResult{Trees : res , VisitedCount :
int(atomic.LoadInt64(&counter))};
}
}

```

Berdasarkan potongan kode diatas, DFS menggunakan pendekatan rekursif EnumerateDFS dengan memoisasi di sync.Map memory. Setiap kali memasuki sebuah elemen, touch() dilakukan untuk menambah counter. Adanya stack[name] mencegah infinite loop pada siklus, sementara memory.Load dan memory.Store dilakukan untuk menghindari perhitungan ulang subtree. Setelah perhitungan rekursif subtree untuk setiap pasangan parent, dikumpulkan local_res pada setiap goroutine, lalu menggabungkannya ke res (utama) dengan mutex.Lock(). Hanya pasangan (l , r) dengan tier yang lebih rendah yang akan dilanjutkan, dan dibatasi dengan sebanyak max_recipe resep pertama yang ditemukan dikembalikan. Sistem stack di sini memastikan traversal DFS, sedangkan memory berfungsi sebagai cache untuk subtree. Mutex dilakukan untuk menjamin hanya ada satu goroutine saja yang boleh memegang kunci dalam satu waktu.

4. BDR & EnumerateTopBDR & EnumerateBotBDR (Bidirectional)

```

func EnumerateTopBDR(gallery *Gallery , name string , mid_tier int ,
memory map[string][]*RecipeNode) []*RecipeNode {
    touch();
    tier := GetTier(gallery , name);
    if (tier <= mid_tier) {
        return []*RecipeNode{{Name : name}};
    } else if res , check := memory[name] ; check {
        return res;
    } else {
        var res []*RecipeNode;
        element := gallery.GalleryName[name];
        for _ , parent := range element.Parents {
            touch();
            l := parent[0];
            r := parent[1];
            if (GetTier(gallery , l) < tier && GetTier(gallery , r) <
tier) {
                pl := EnumerateTopBDR(gallery , l , mid_tier , memory);
                pr := EnumerateTopBDR(gallery , r , mid_tier , memory);
                for _ , ll := range pl {
                    for _ , rr := range pr {
                        res = append(res , &RecipeNode{Name : name ,
Parents : []*RecipeNode{ll , rr}});
                    }
                }
            }
            memory[name] = res;
            return res;
        }
    }
}

func EnumerateBotBDR(gallery *Gallery, name string, memory
map[string][]*RecipeNode) []*RecipeNode {
    touch();
    if res , check := memory[name] ; check {
        return res;
    } else {
        element := gallery.GalleryName[name];
        if (element == nil || element.Tier == 0 || len(element.Parents)
== 0) {
            res := []*RecipeNode{{Name : name}};
        }
    }
}

```

```

        memory[name] = res;
        return res;
    } else {
        var res []*RecipeNode
        for _, parent := range element.Parents {
            touch();
            l := parent[0];
            r := parent[1];
            if (GetTier(gallery, l) < element.Tier &&
GetTier(gallery, r) < element.Tier) {
                pl := EnumerateBotBDR(gallery, l, memory);
                pr := EnumerateBotBDR(gallery, r, memory);
                for _, l := range pl {
                    for _, r := range pr {
                        res = append(res, &RecipeNode{Name : name
, Parents : []*RecipeNode{l, r}}));
                    }
                }
            }
            memory[name] = res;
            return res;
        }
    }
}

func BDR(gallery *Gallery, target string, max_recipe int)
AlgorithmResult {
    if (GetTier(gallery, target) == 0) {
        touch();
        node := &RecipeNode{Name : target};
        return AlgorithmResult{Trees : []*RecipeNode{node} ,
VisitedCount : int(atomic.LoadInt64(&counter))};
    } else {
        if (max_recipe == 0) {
            max_recipe = int(^uint(0) >> 1);
        }
        mid_tier := GetMidTier(gallery, target);
        memory_top := make(map[string][]*RecipeNode);
        EnumerateTopBDR(gallery, target, mid_tier, memory_top);
        mnext := memory_top[target];
        group := make(map[string][]*RecipeNode);
        for _, next := range mnext {

```

```

var Collecting func(*RecipeNode);
Collecting = func(node *RecipeNode) {
    if (len(node.Parents) == 0) {
        if (GetTier(gallery, node.Name) >= mid_tier) {
            group[node.Name] = append(group[node.Name] ,
next);
        }
    } else {
        for _, parent := range node.Parents {
            Collecting(parent);
        }
    }
}
Collecting(next)
}
var (
    wg    sync.WaitGroup;
    res   []*RecipeNode;
    mutex sync.Mutex;
    signature_tree = make(map[string]struct{});
)
for name, parts := range group {
    wg.Add(1);
    go func() {
        defer wg.Done();
        memory_bot := make(map[string][]*RecipeNode);
        trees_bot := EnumerateBotBDR(gallery, name,
memory_bot);
        var local_res []*RecipeNode;
        for _, pt := range parts {
            for _, bt := range trees_bot {
                touch();
                clone_top, clone_map := CloneTreeMap(pt);
                var leaf *RecipeNode;
                var FindLeaf func(*RecipeNode)
                FindLeaf = func(node *RecipeNode) {
                    if (leaf != nil) {
                        return;
                    } else if (node.Name == name &&
len(node.Parents) == 0) {
                        leaf = node;
                    } else {
                        for _, parent := range node.Parents {

```

```

                FindLeaf(parent);
            }
        }
        FindLeaf(pt);
        clone_bot , _ := CloneTreeMap(bt);
        clone_map[leaf].Parents = clone_bot.PARENTS;
        signature := SignatureTree(clone_top);
        mutex.Lock();
        if _ , check := signature_tree[signature] ;
    !check {
        signature_tree[signature] = struct{}{};
        local_res = append(local_res , clone_top);
        if (len(local_res) >= max_recipe) {
            mutex.Unlock();
            goto flush;
        }
    }
    mutex.Unlock();
}
flush :
    mutex.Lock();
    res = append(res , local_res...);
    mutex.Unlock();
}();
}
wg.Wait();
if (len(res) > max_recipe) {
    res = res[:max_recipe];
}
return AlgorithmResult{Trees : res , VisitedCount :
int(atomic.LoadInt64(&counter))};
}
}

```

Berdasarkan potongan kode diatas, BDR memotong pencarian menjadi dua arah berdasarkan mid_tier yang telah dihitung terlebih dahulu dengan GetMidTier. Pertama, EnumerateTopBDR akan menjelajah gallery secara top-down dari target sampai tier-nya kurang dari mid_tier, lalu semua subtree top disimpan di memory_top. Dari setiap tree top, fungsi Collecting mengekstrak node-node leaf di level meet-point dan

mengelompokkan tree top berdasarkan nama leaf tersebut (group[name]). Lalu untuk setiap group dilakukan goroutine dengan memanggil EnumerateBotBDR yang akan menjelajah gallery secara bottom-up dari tiap name, lalu meng-clone kedua pohon dengan CloneTreeMap, menyatukan keduanya pada leaf yang sama, dan mengabaikan duplikasi dengan menggunakan SignatureTree dan map signature_tree. Selama proses, touch() menghitung kunjungan node, dan memory mencegah duplikasi setiap enumerasi node. Hasilnya adalah semua gabungan top dan bottom yang valid sampai max_recipe. Mutex dilakukan untuk menjamin hanya ada satu goroutine saja yang boleh memegang kunci dalam satu waktu.

5. Multithreading (Enable & Disable)

```
func EnableMultithreading() {
    runtime.GOMAXPROCS(max(runtime.NumCPU()/2, runtime.NumCPU()-4));
}

func DisableMultithreading() {
    runtime.GOMAXPROCS(1);
}
```

Berdasarkan potongan kode diatas, multithreading merupakan konsep untuk menyediakan banyak thread CPU untuk suatu proses, sehingga task yang harusnya dikerjakan oleh 1 thread dapat dibagi ke banyak thread lainnya. Pada implementasi EnableMultithreading, dilakukan penyisaan sebanyak minimal antara 4 thread CPU atau separuh dari keseluruhan untuk keperluan di luar program, namun sisanya digunakan secara aktif untuk mempercepat proses pencarian pada program. Pada implementasi DisableMultithreading, thread CPU yang digunakan dikembalikan seperti default yaitu 1 saja. Dengan adanya multithreading, implementasi counter visited node sedikit berbeda karena harus dipandang sebagai atomic dengan detail pada Bab 4.1.3. Lalu, diperlukan pula Mutex untuk menjaga integritas antar goroutine dalam memegang suatu kunci yang hanya boleh 1 goroutine saja dalam 1 waktu yang sama.

4.1.3 Fungsi Dan Prosedur Pendukung

```
func IsBase(name string) bool {
    _, check := base_element[name];
```

```

        return check;
    }

func FindPointer(root *RecipeNode , name string) *RecipeNode {
    if (root == nil) {
        return nil;
    } else if (root.Name == name && len(root.Parents) == 0) {
        return root;
    } else {
        for _, parent := range root.Parents {
            if node := FindPointer(parent , name) ; node != nil {
                return node;
            }
        }
        return nil;
    }
}

func IsExpandable(element *Element) bool {
    return element != nil && element.Tier > 0 && len(element.Parents) > 0;
}

func GetMidTier(gallery *Gallery , target string) int {
    return GetTier(gallery , target) / 2;
}

func CloneNode(node *RecipeNode) *RecipeNode {
    if (node == nil) {
        return nil;
    } else {
        parents := make([]*RecipeNode , len(node.Parents));
        for idx , cp := range node.Parents {
            parents[idx] = CloneNode(cp);
        }
        return &RecipeNode{Name : node.Name , Parents : parents};
    }
}

func CloneSlice(src []*RecipeNode) []*RecipeNode {
    cp := make([]*RecipeNode , len(src));
    for idx , node := range src {
        cp[idx] = CloneNode(node);
    }
}

```

```

        return cp;
    }

func SignatureTree(node *RecipeNode) string {
    if (node == nil) {
        return "";
    } else if len(node.Parents) == 0 {
        return node.Name;
    } else {
        l := SignatureTree(node.Parents[0]);
        r := SignatureTree(node.Parents[1]);
        return node.Name + "(" + l + "," + r + ")";
    }
}

func CloneTreeMap(original *RecipeNode) (*RecipeNode ,
map[*RecipeNode]*RecipeNode) {
    clone_map := make(map[*RecipeNode]*RecipeNode);
    var CloneRecursive func(*RecipeNode) *RecipeNode;
    CloneRecursive = func(node *RecipeNode) *RecipeNode {
        if (node == nil) {
            return nil;
        } else {
            clone := &RecipeNode{Name : node.Name};
            clone_map[node] = clone;
            if (len(node.Parents) == 2) {
                clone.Parents =
                []*RecipeNode{CloneRecursive(node.Parents[0]) ,
CloneRecursive(node.Parents[1])};
            }
            return clone;
        }
    }
    return CloneRecursive(original) , clone_map;
}

func (num *RecipeNode) Marshal() ([]byte , error) {
    return json.MarshalIndent(num , "" , " ");
}

func Transform(array []any) [][]string {
    var res [][]string;
    for _ , pairs := range array {

```

```

        var pair []string;
        for _, s := range pairs.([]any) {
            pair = append(pair, s.(string));
        }
        res = append(res, pair);
    }
    return res;
}

func GetTier(gallery *Gallery, name string) int {
    if element, check := gallery.GalleryName[name]; check {
        return element.Tier;
    } else {
        return 0;
    }
}

func CalculateTier(name string, gallery *Gallery, visited
map[string]bool) int {
    if (IsBase(name)) {
        return 0;
    } else {
        element := gallery.GalleryName[name];
        if (element == nil) {
            return 1;
        } else if (element.Tier >= 0) {
            return element.Tier;
        } else if (visited[name]) {
            return 1;
        } else {
            best := 0;
            visited[name] = true;
            for _, p := range element.Parents {
                t1 := CalculateTier(p[0], gallery, visited);
                t2 := CalculateTier(p[1], gallery, visited);
                if t := max(t1, t2) + 1; t > best {
                    best = t;
                }
            }
            visited[name] = false;
            element.Tier = best;
        }
    }
    return best;
}

```

```

        }
    }

func (gallery *Gallery) GetAllNames() []string {
    names := make([]string, 0, len(gallery.GalleryName));
    for name := range gallery.GalleryName {
        names = append(names, name);
    }
    return names;
}

var counter int64;
func touch() {
    atomic.AddInt64(&counter, 1);
}

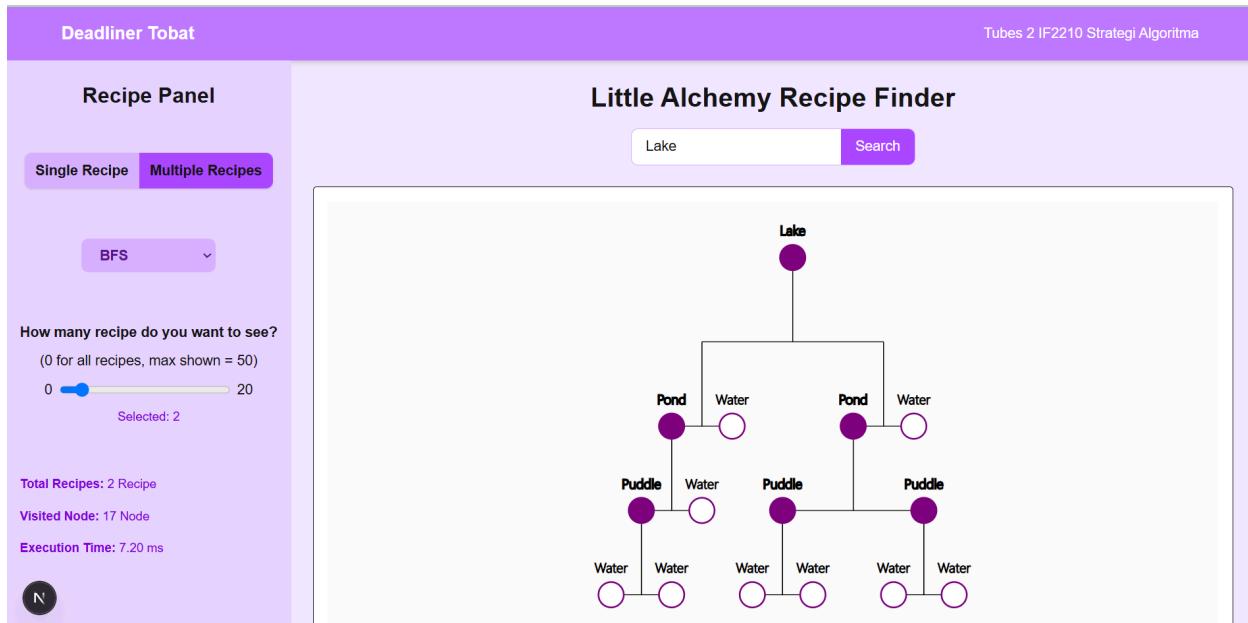
```

Berdasarkan potongan kode diatas, kami membuat beberapa fungsi dan prosedur tambahan untuk memudahkan utilitas program. Pada IsBase, dilakukan pengecekan apakah sebuah elemen merupakan base elemen atau bukan. Pada FindPointer, dilakukan pencarian terhadap pointer ke node leaf yang dicari sesuai input node utama dan name yang dicari. Pada IsExpandable, dilakukan pengecekan apakah sebuah elemen masih dapat diperluas pencarinya atau tidak. Pada GetMidTier, dilakukan penghitungan MidTier dari elemen yang dicari untuk digunakan dalam Algoritma Bidirectional. Pada CloneNode, dilakukan salinan deep-copy terhadap suatu node beserta seluruh subtree-nya juga. Pada CloneSlice, dilakukan proses clone dari suatu slice atas kumpulan RecipeNode yang masing-masingnya di-clone dengan CloneNode. Pada SignatureTree, dilakukan pembuatan identitas unik berupa signature dari suatu node menjadi sebuah string atas Name dan Parents-nya. Pada CloneTreeMap, dilakukan proses clone terhadap suatu RecipeNode dengan mengembalikan map yang memetakan node asal menuju node hasil clone. Pada Marshal, dilakukan untuk memudahkan serialisasi satu pohon RecipeNode ke JSON dengan tambahan indentasi. Pada Transform, dilakukan pengkonversian data mentah hasil decoding JSON berupa array of any menjadi matrix of string. Pada GetTier, dilakukan pengembalian nilai tier sebuah elemen jika terdefinisi di GalleryName atau placeholder nilai tier 0 jika tidak terdefinisi. Pada CalculateTier, dilakukan penghitungan tier elemen sebagai antisipasi jika belum terdefinisi dengan baik dari data di JSON dengan mencari depth dari base element. Pada GetAllNames,

dilakukan pengumpulan terhadap seluruh slice nama dari semua elemen yang ada di `GalleryName`. Serta, pada touch, dilakukan penambahan counter secara atomic untuk menghitung `VisitedNodeCount` dengan tipe data `int64` counter.

4.2 Antarmuka dan Penggunaan Aplikasi

4.2.1 Antarmuka Aplikasi



Pada aplikasi ini, terdapat sidebar untuk memasukkan permintaan algoritma dan banyaknya resep yang ingin dicari. Terdapat toggle untuk memilih apakah ingin single recipe atau multiple recipe. Di bawah toggle, terdapat dropdown untuk memilih algoritma apa yang ingin digunakan. Jika pengguna menekan toggle multiple recipes, terdapat pilihan dalam bentuk *slider* berapa banyak resep yang ingin dicari. Di tengah atas terdapat *search bar* untuk memasukkan elemen apa yang ingin dicari oleh pengguna. Setelah menekan tombol “Search”, hasil tree akan ditampilkan pada area berwarna putih di bawah *search bar* dan akan ditampilkan juga berapa banyak total resep, node yang dikunjungi, serta waktu eksekusi program pada sidebar di paling bawah.

4.2.2 Fitur-fitur yang Disediakan Aplikasi

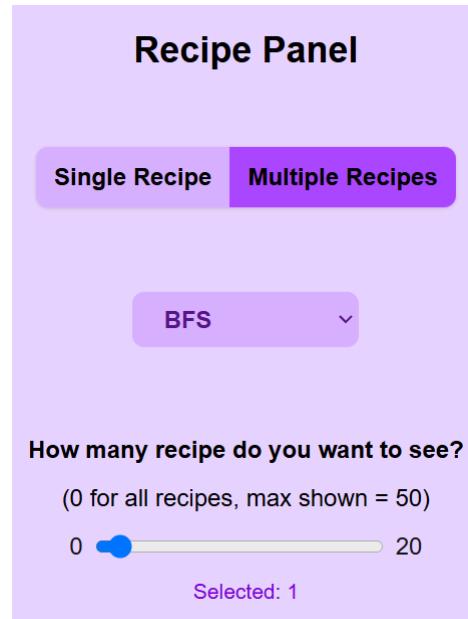
Aplikasi ini memiliki beberapa fitur utama yang dirancang untuk membantu pengguna menemukan *recipe* dari elemen yang akan dicari dalam berbagai algoritma pencarian. Fitur-fitur tersebut antara lain:

- Pemilihan Algoritma (BFS/DFS/Bidirectional)



Aplikasi memberikan fitur dropdown untuk pengguna memilih algoritma apa yang akan digunakan untuk mencari elemen. Terdapat tiga algoritma yang tersedia pada aplikasi ini, yaitu algoritma BFS, DFS, serta Bidirectional.

- Pemilihan Banyaknya Recipe



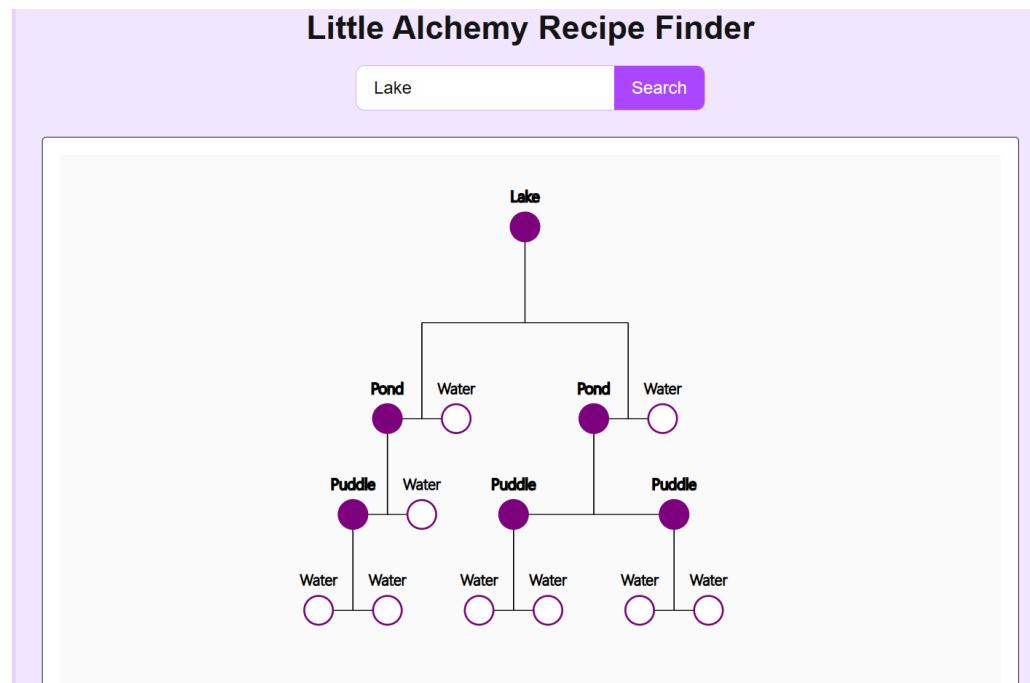
Dalam sidebar, terdapat Recipe Panel untuk memasukkan permintaan algoritma yang akan digunakan untuk mencari elemen serta banyaknya resep yang ingin diketahui dari elemen tersebut. Pengguna juga bisa memasukkan 0 recipe untuk melakukan pencarian terhadap semua resep yang ada dengan batas maksimum recipe yang dicari adalah 50.

- Pencarian Elemen



Pengguna dapat memasukkan nama elemen pada *search bar* yang disediakan di bawah judul aplikasi “Little Alchemy Recipe Finder”. Setelah memasukkan nama elemen yang ingin dicari, pengguna dapat menekan tombol “Search” untuk kemudian meminta program mencari *recipe* yang sesuai dengan algoritma dan banyaknya *recipe* yang diminta pengguna sebelumnya.

- Visualisasi Recipe Tree



Setelah pengguna menekan tombol “Search”, program akan memproses pencarian *recipe* dari elemen yang dimasukkan oleh pengguna. Kemudian, hasil tree *recipe* akan ditampilkan seperti pada gambar di atas.

- Hasil Total Recipe, Visited Nodes, dan Execution Time

Total Recipes: 6 Recipe
Visited Node: 69 Node
Execution Time: 7.60 ms

Selain gambar *tree recipe*, aplikasi juga akan menampilkan total resep, banyaknya node yang dikunjungi, serta lama waktu eksekusi program dalam mencari *recipe* elemen tersebut.

4.2.3 Tata Cara Penggunaan Aplikasi

Aplikasi ini dirancang agar mudah digunakan oleh pengguna dalam menelusuri *resep* pembuatan suatu elemen berdasarkan kombinasi elemen lainnya. Antarmuka pengguna terdiri atas beberapa komponen yang intuitif dan interaktif. Langkah-langkah penggunaan aplikasi adalah sebagai berikut:

1. Memasukkan Nama Elemen

Pengguna mengetikkan nama elemen target yang ingin dicari pada kolom input yang tersedia di bagian kiri aplikasi. Elemen yang dimasukkan harus sesuai dengan nama elemen yang terdapat dalam basis data aplikasi.

2. Memilih Algoritma Pencarian

Pengguna dapat memilih salah satu algoritma pencarian yang tersedia, yaitu:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Bidirectional Search

3. Pemilihan algoritma dilakukan melalui dropdown menu yang terletak di bawah kolom input.

4. Menentukan Jumlah Maksimal Resep yang Dicari

Aplikasi menyediakan opsi untuk membatasi jumlah *recipe* yang dicari, dalam bentuk *slider bar*. Hal ini bertujuan untuk menghindari visualisasi *tree* yang terlalu kompleks dan mempercepat proses pencarian.

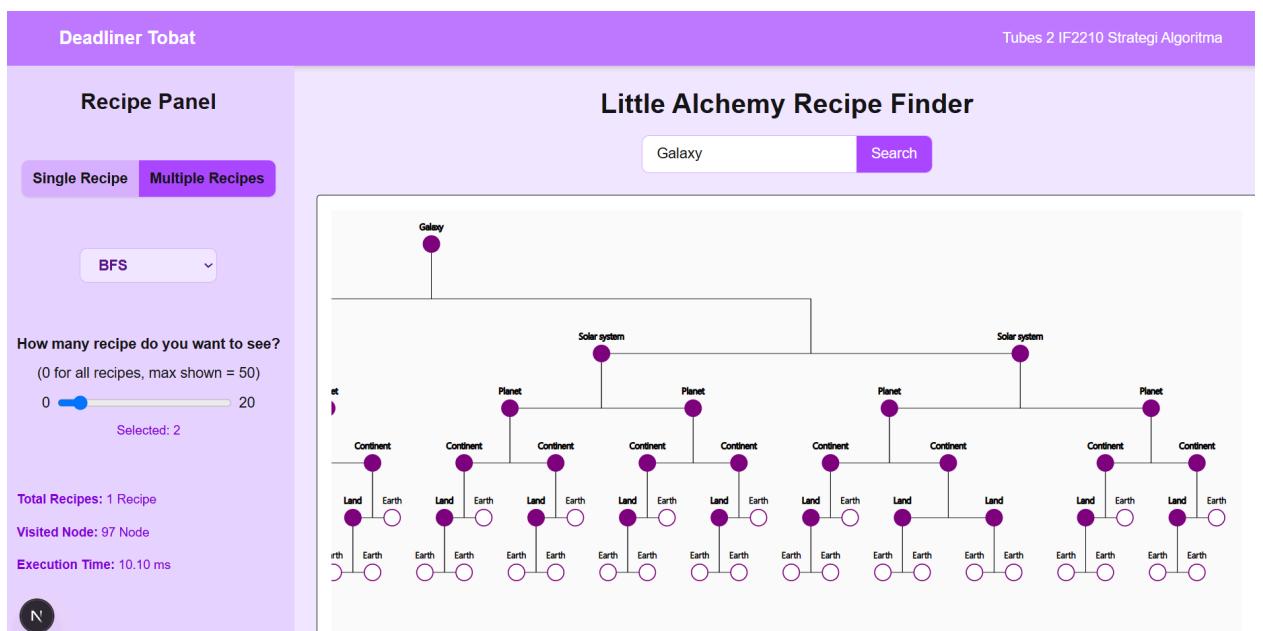
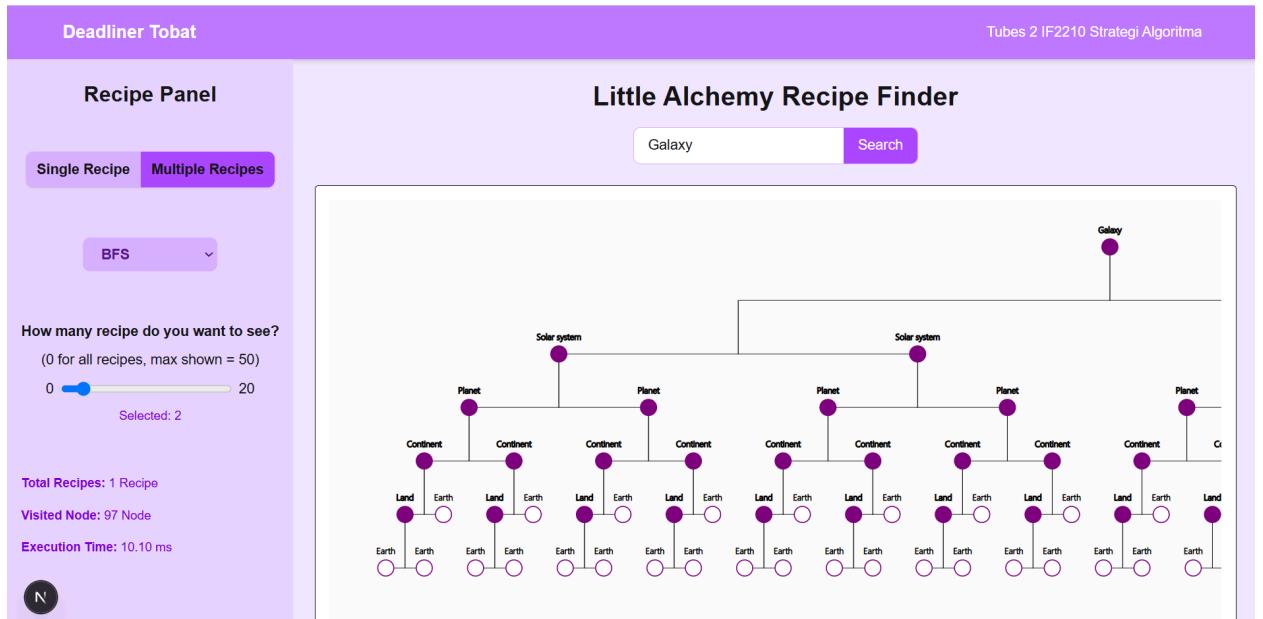
5. Menjalankan Proses Pencarian

Setelah semua input diisi, pengguna dapat menekan tombol “Search” untuk memulai proses pencarian. Aplikasi akan mengirim permintaan ke server dan menampilkan

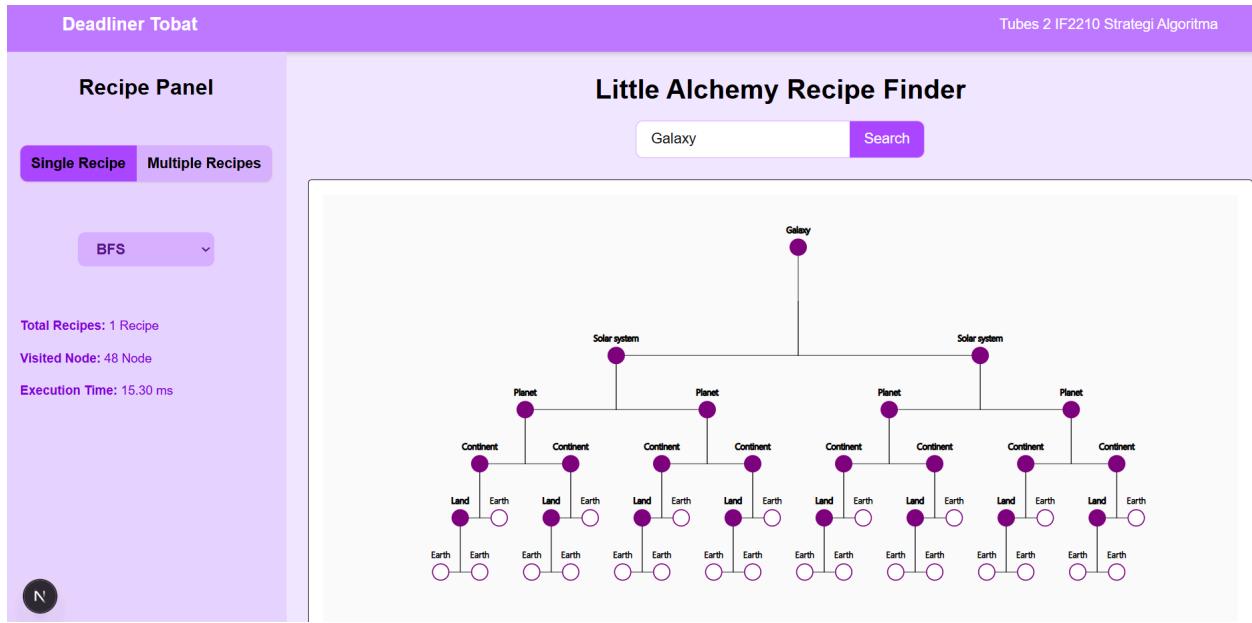
hasil berdasarkan algoritma yang dipilih.

4.3 Hasil Pengujian

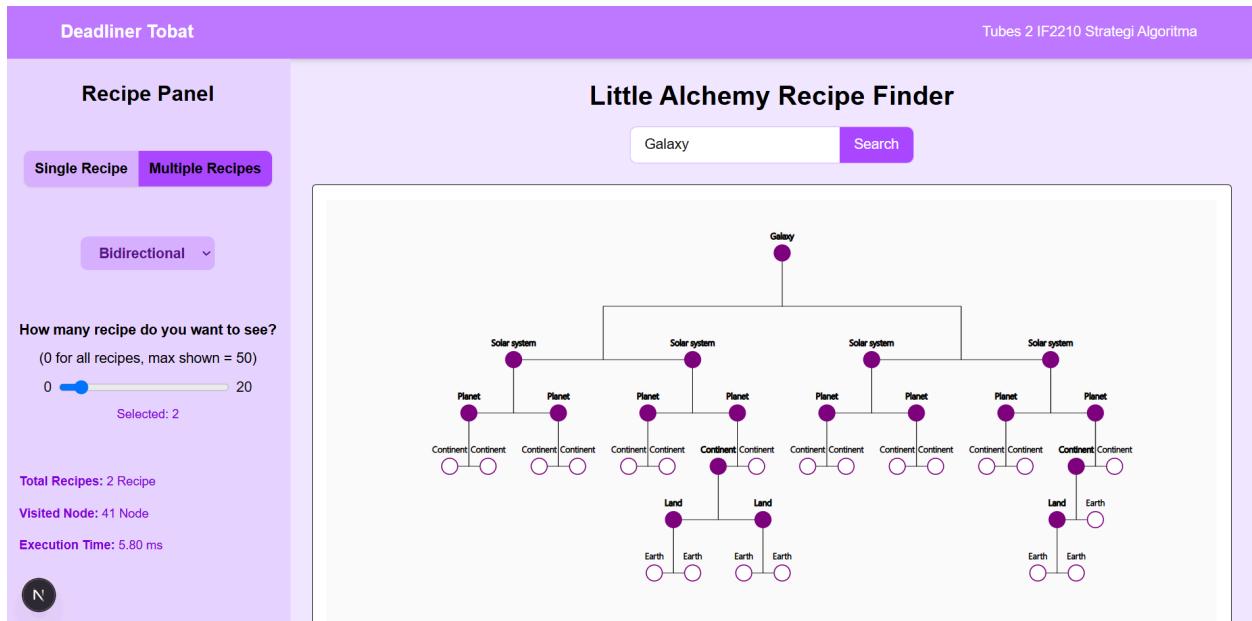
- Mencari Elemen Galaxy dengan BFS Multi Recipes = 2



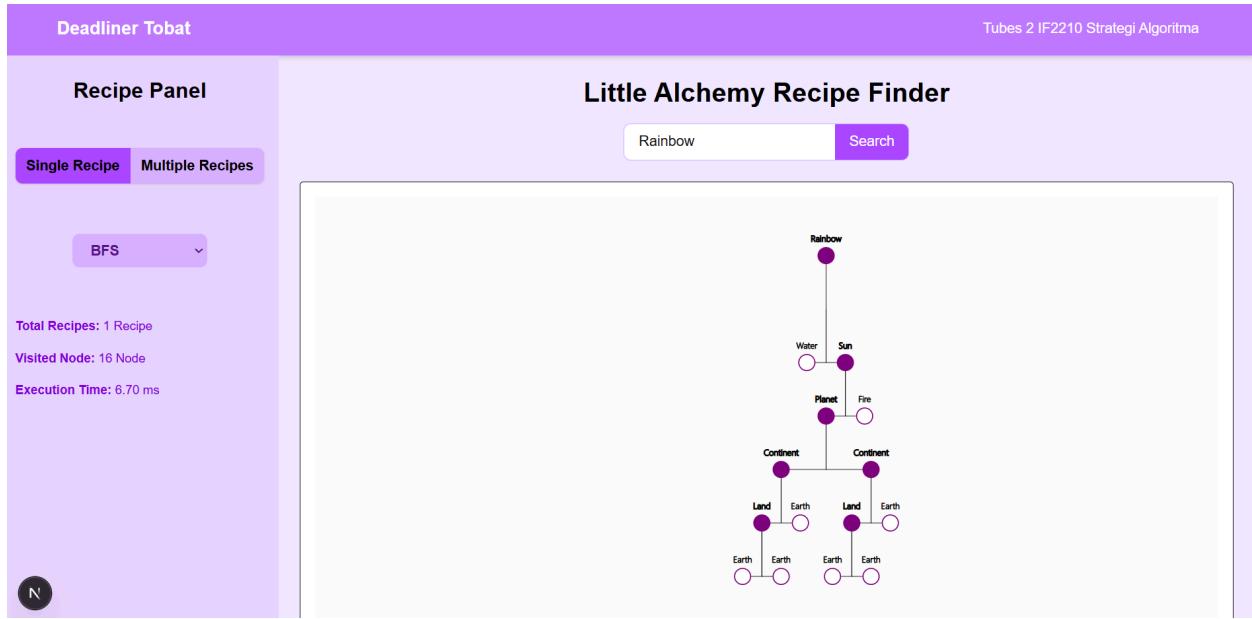
- Mencari Elemen Galaxy dengan BFS Single Recipes



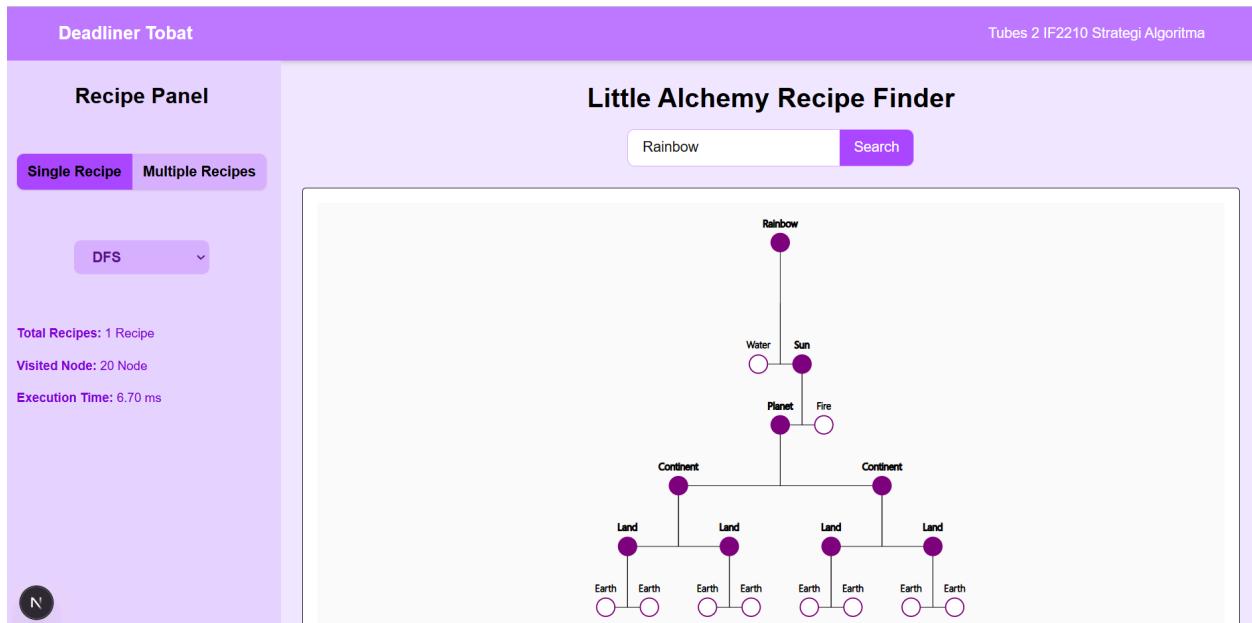
- Mencari Elemen Galaxy dengan Bidirectional Multiple Recipes = 2



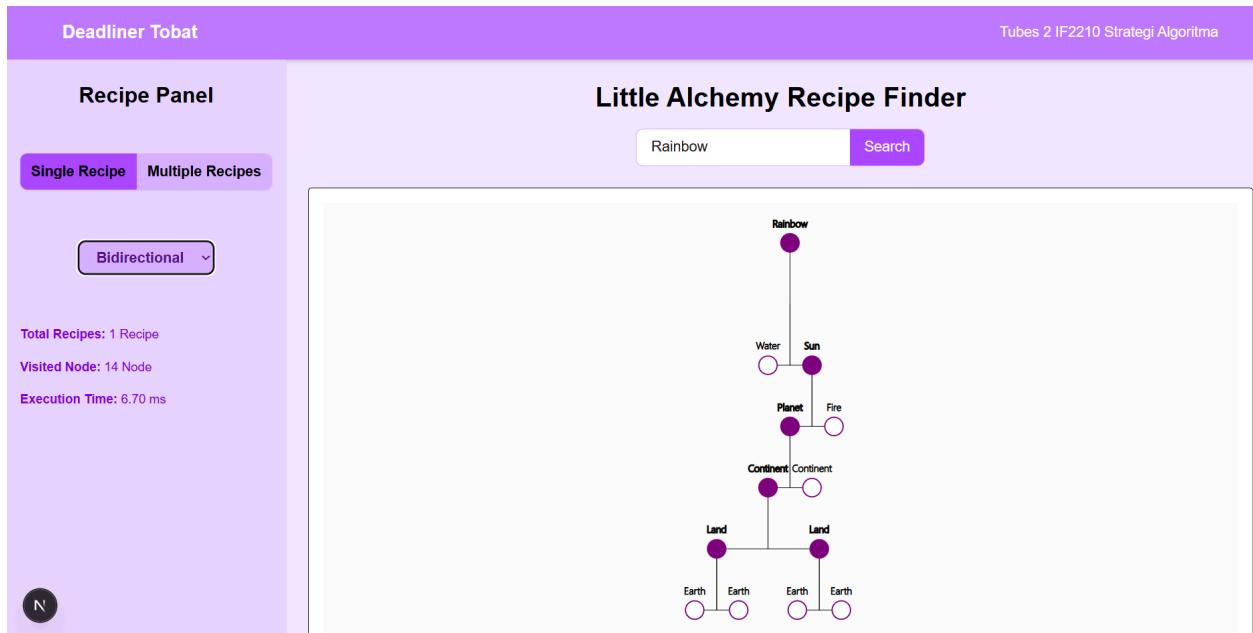
- Mencari Elemen Rainbow dengan BFS Single Recipes



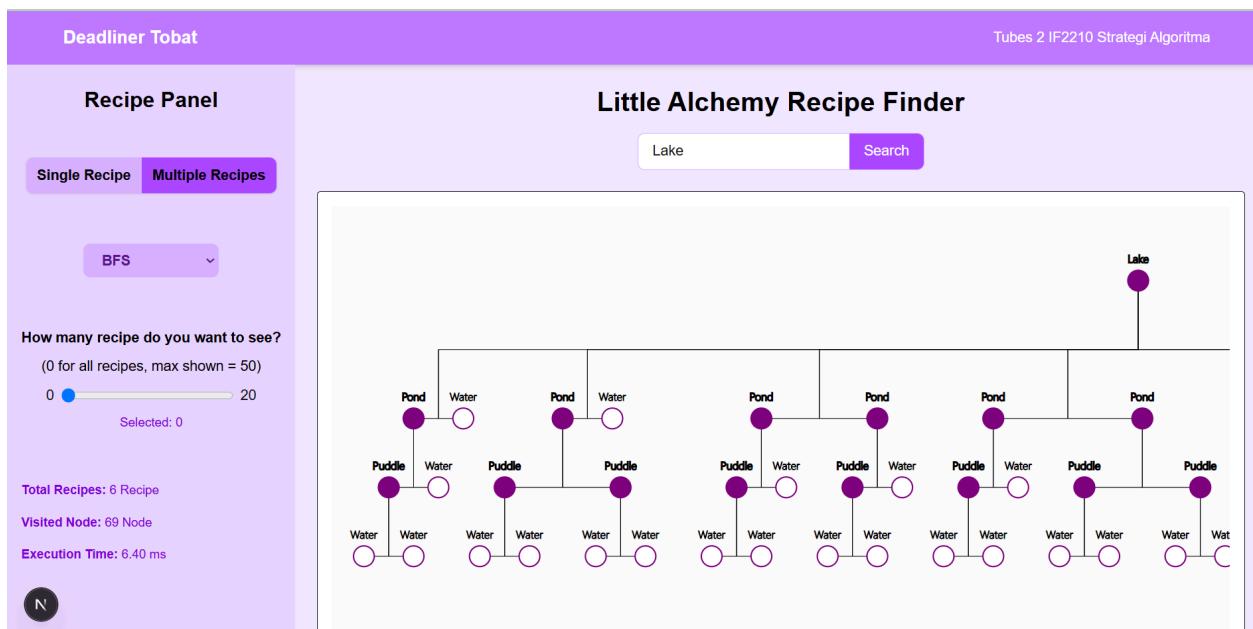
- Mencari Elemen Rainbow dengan DFS Single Recipes



- Mencari Elemen Rainbow dengan Bidirectional Single Recipes

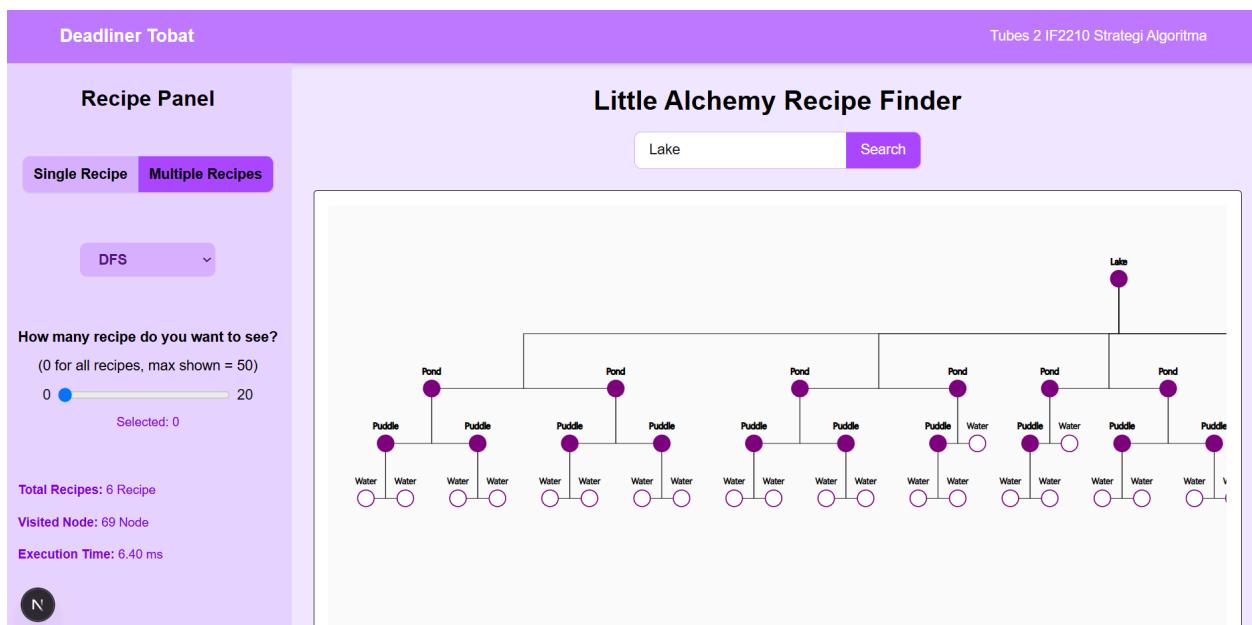


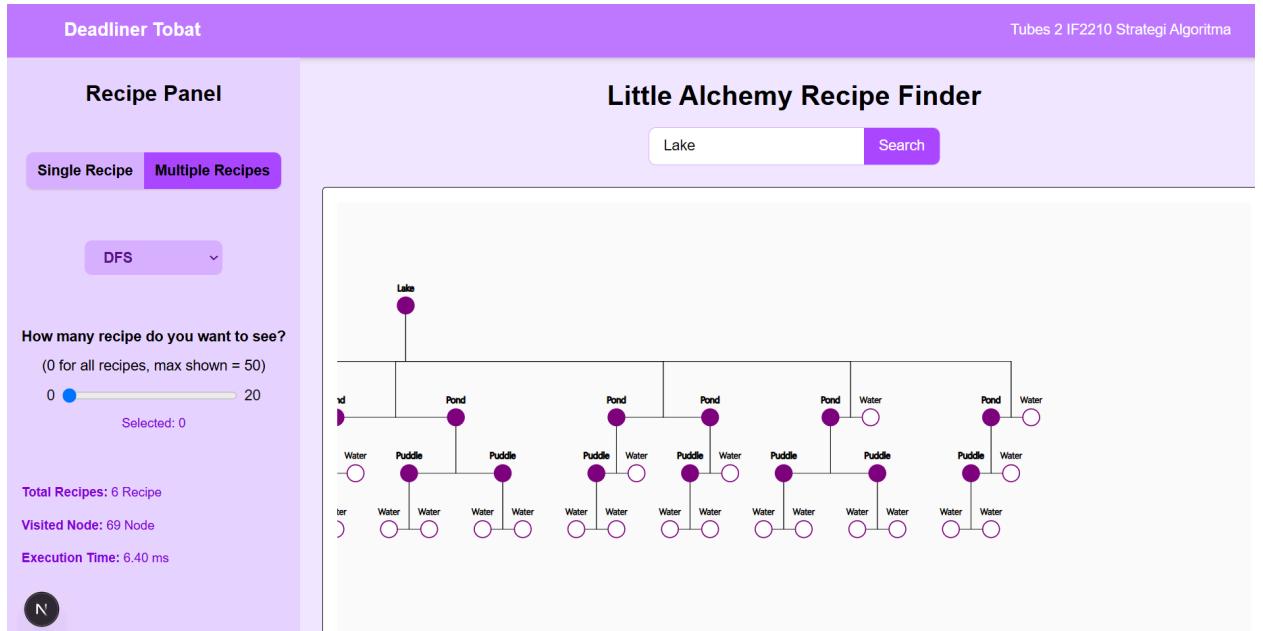
- Mencari Elemen Lake dengan BFS Multiple Recipes = 0 (All Recipes)



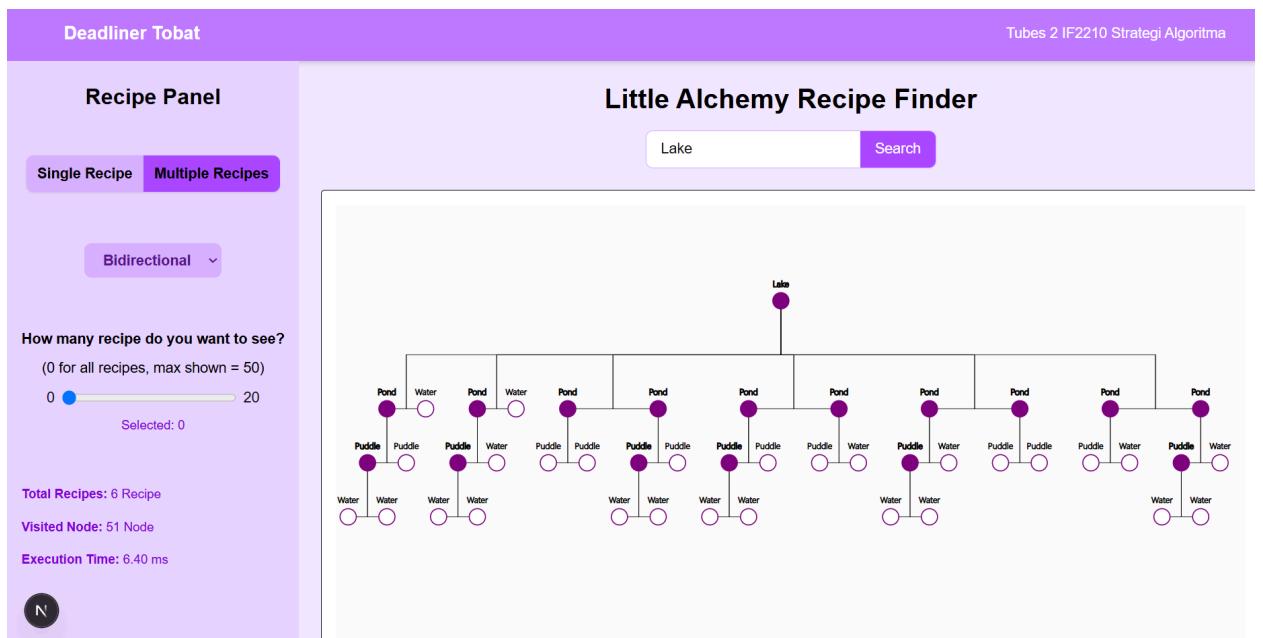


- Mencari Elemen Lake dengan DFS Multiple Recipes = 0 (All Recipes)





- Mencari Elemen Lake dengan Bidirectional Multiple Recipes = 0 (All Recipes)



4.4 Analisis Hasil Pengujian

Berdasarkan hasil pengujian yang dilakukan terhadap berbagai elemen seperti *Galaxy*, *Lake*, dan *Rainbow* dengan menggunakan algoritma Breadth First Search (BFS), Depth First Search (DFS), dan Bidirectional Search, dapat disimpulkan bahwa sistem telah berhasil menampilkan hasil pencarian secara fungsional dan visual sesuai dengan spesifikasi. Pengujian dilakukan dalam dua

mode utama, yaitu Single Recipe dan Multiple Recipes, dengan variasi jumlah maksimal resep yang dicari.

Pada mode *Multiple Recipes*, sistem mampu menampilkan beberapa jalur pembentukan elemen target yang valid. Misalnya, saat mencari elemen *Lake* menggunakan BFS dan DFS, aplikasi menunjukkan beberapa tree berbeda yang memiliki struktur akar yang sama (*Lake*) namun berasal dari kombinasi bahan yang berbeda-beda. Hal ini menunjukkan bahwa sistem berhasil melakukan eksplorasi penuh terhadap semua jalur yang memungkinkan hingga menghasilkan *Lake*.

Pada mode *Single Recipe*, sistem hanya mencari satu jalur pembentukan tercepat, dan hasil visualisasi menunjukkan struktur pohon yang lebih ringkas dibandingkan mode *Multiple Recipes*. Contohnya pada pencarian elemen *Rainbow* dengan BFS dan DFS, hanya satu jalur yang divisualisasikan dan jumlah node yang dikunjungi jauh lebih sedikit. Hal ini menunjukkan efisiensi algoritma ketika hanya diminta satu solusi, tanpa perlu menjelajah semua kemungkinan.

Secara umum, keseluruhan fitur mulai dari input elemen, pemilihan algoritma, pembatasan jumlah resep, hingga visualisasi pohon dan metrik performa telah berjalan dengan baik. Untuk lebih detail, algoritma DFS akan sangat efektif untuk mencari 1 rute atau setidaknya memastikan apakah ada rute menuju suatu titik tujuan dari suatu titik awal, algoritma BFS akan sangat efektif untuk mencari 1 rute terdekat meskipun agak lama karena harus looping queue yang sangat lama, dan algoritma Bidirectional akan lebih efektif daripada algoritma BFS karena secara umum cara kerjanya serupa hanya saja dari kedua arah sehingga kompleksitasnya lebih rendah.

BAB 5 Kesimpulan, Saran, dan Refleksi

5.1 Kesimpulan

Dari tugas besar ini, kami mempelajari hal baru seperti web development dan juga bahasa baru yaitu Golang. Kami juga memperkuat pengetahuan kami mengenai materi mata kuliah Strategi Algoritma, yaitu materi Breadth First Search (BFS) dan Depth First Search (DFS), serta menambah pengetahuan terkait strategi Bidirectional. Selain itu, kami melatih kemampuan kerjasama dan juga komunikasi antar anggota tim yang tentunya krusial untuk masa depan.

5.2 Saran

Untuk aplikasi semacam ini, fitur bonus Live Traversal bisa dikembangkan lebih lanjut dengan WebSocket atau SSE agar pengguna bisa melihat traversal node secara *real-time*, meningkatkan *experience* aplikasi ini. Pemisahan repository pengembangan frontend dan backend juga bisa dilakukan sejak awal agar proses deployment lebih efisien dan tidak saling bergantung. Ini juga akan memudahkan penggunaan layanan deployment seperti Vercel dan Railway.

5.3 Refleksi

Nama kelompok kami, DeadlinerTobat, adalah bentuk harapan dari kami yang merasa bahwa kami merupakan sekumpulan orang *deadliner* agar tidak mengumpulkan tugas besar ini mendekati waktu *deadline*. Akan tetapi, kami masih gagal melakukannya. Hal ini akan menjadi pelajaran bagi kami kedepannya untuk menyusun jadwal dan mengalokasikan waktu kami dengan lebih baik.

Lampiran

1. Pranala Github : https://github.com/RNXFreeze/Tubes2_DeadlinerTobat/
2. Pranala Video Youtube : <https://youtu.be/PtW3NIDvu98>
3. Tabel Checklist :

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 <i>Bidirectional</i> .	✓	
9	Membuat bonus <i>Live Update</i> .		✓
10	Aplikasi di- <i>containerize</i> dengan Docker.		✓
11	Aplikasi di- <i>deploy</i> dan dapat diakses melalui internet.		✓

Daftar Pustaka

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf) (Diakses 13 Mei 2025)

<https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/> (Diakses 13 Mei 2025)

<https://www.geeksforgeeks.org/bidirectional-search/> (Diakses 13 Mei 2025)