

# **Laporan Tugas Besar 2 IF2211 Strategi Algoritma**

## **Pencarian Recipe Alchemy**



**Disusun Oleh:**  
**Kelompok 5 - SOS**

Muhammad Naufal Rayhannida	10123006
Ivant Samuel Silaban	13523129
Muhammad Rizain Firdaus	13523164

**Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung**

**2025**

# DAFTAR ISI

<b>BAB 1: Deskripsi Tugas</b>	<b>4</b>
1. Latar Belakang	4
<b>BAB 2: Landasan Teori</b>	<b>6</b>
1. Dasar Teori	6
1.1. Breadth First Search (BFS)	6
1.2. Depth First Search (DFS)	6
1.3. Bidirectional Search	7
2. Mengenai Website	8
<b>BAB 3: Analisis Pemecahan Masalah</b>	<b>9</b>
1. Langkah-langkah Pemecahan Masalah	9
1. Pemahaman Masalah	9
2. Pengumpulan Data	9
3. Pemodelan Masalah	9
4. Implementasi Algoritma	9
5. Validasi Tier	10
6. Visualisasi dan Pengukuran	10
2. Proses Pemetaan Masalah Menjadi Elemen-elemen Algoritma DFS dan BFS	10
1. Representasi Graf	10
2. Algoritma BFS	10
3. Algoritma DFS	11
4. Algoritma Bidirectional (Bonus)	11
5. Pencarian Multi Recipes	11
3. Fitur Fungsional dan Arsitektur Aplikasi Web	11
4. Contoh Ilustrasi Kasus	12
1. Pencarian dengan BFS:	12
2. Pencarian dengan DFS:	13
3. Pencarian Multi Recipe	13
<b>BAB 4: Implementasi dan Pengujian</b>	<b>15</b>
4.1 Spesifikasi Teknis Program	15
4.1.1 Struktur Data	15
1. Combination	15
2. Node	15
3. RecipePath	16
4. Map Kombinasi dan Tier	16
5. Counter	16
4.1.2 Fungsi dan Prosedur	16
1. LoadCombinations(filename string) error	16
2. isBasic(element string) bool	17

3. FindRecipeBFS(target string) *Node	18
4. FindRecipeDFS(target string, visited map[string]bool) *Node	21
5. FindRecipeBidirectional(target string) *Node	22
6. FindMultipleRecipes(target string, maxCount int) []*Node	26
7. FindRecipeBidirectional(target string, startElement string) *Node:	32
8. Fungsi Pembantu:	35
4.2 Penjelasan Tata Cara Penggunaan Program	36
4.2.1 Cara Menjalankan Program secara Lokal	36
4.2.2 Cara Menjalankan Program secara Online	37
4.2.1 Cara Menjalankan Program dengan Docker	37
4.2.4 Cara Penggunaan Program	37
4.2.4.1 Tampilan Awal	37
4.2.4.2 Tampilan Single Recipe	38
4.2.4.3 Tampilan Multi Recipe	39
4.3 Hasil Pengujian	43
4.3.1 Hasil Pengujian untuk Single Recipe	43
4.3.2 Hasil Pengujian untuk Multi Recipe:	56
4.4 Analisis Hasil Pengujian	64
4.4.1 Analisis Efisiensi Algoritma	65
4.4.2 Analisis Waktu Eksekusi	67
4.4.2 Keterbatasan dan Rekomendasi	67
<b>BAB 5: Kesimpulan dan Saran</b>	<b>69</b>
5.1 Kesimpulan	69
5.2 Saran	69
5.3 Refleksi	69
<b>LAMPIRAN</b>	<b>70</b>
Link Repository : <a href="https://github.com/ivant8k/Tubes2_SOS">https://github.com/ivant8k/Tubes2_SOS</a>	70
Link Deploy : <a href="https://alchemix.vercel.app/">https://alchemix.vercel.app/</a>	70
Link Video Youtube : <a href="https://youtu.be/CD9W-c6la3k?si=SZM8Owx-eOhw8rj7">https://youtu.be/CD9W-c6la3k?si=SZM8Owx-eOhw8rj7</a>	70
CHECKLIST:	70
<b>DAFTAR PUSTAKA</b>	<b>71</b>

# BAB 1: Deskripsi Tugas

## 1. Latar Belakang



Gambar 1. Little Alchemy 2

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 2. Elemen dasar pada game 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*

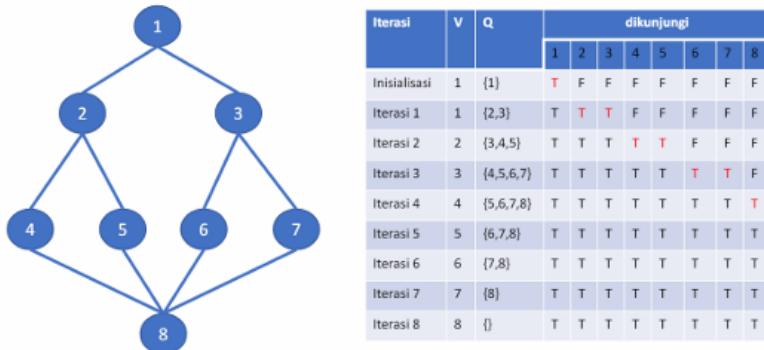
Untuk mendapatkan elemen turunan pemain dapat melakukan combine antara 2 elemen untuk menghasilkan elemen baru. Elemen turunan yang telah didapatkan dapat digunakan kembali oleh pemain untuk membentuk elemen lainnya.

## BAB 2: Landasan Teori

### 1. Dasar Teori

#### 1.1. Breadth First Search (BFS)

BFS merupakan algoritma pencarian yang bekerja dengan prinsip pencarian melebar, yakni mengunjungi semua simpul tetangga terlebih dahulu sebelum melanjutkan ke simpul pada tingkat kedalaman berikutnya. Algoritma ini menggunakan struktur data antrian (queue) untuk menyimpan simpul-simpul yang akan dikunjungi. BFS menjamin ditemukan solusi apabila ada (completeness), dan jika biaya setiap langkah sama, maka solusi yang ditemukan juga optimal. Kompleksitas waktunya adalah  $O(b^d)$ , dan kompleksitas ruangnya juga  $O(b^d)$ , di mana  $b$  adalah branching factor dan  $d$  adalah kedalaman solusi. BFS sangat cocok untuk pencarian rute dengan biaya minimum, seperti pada pencarian jalan terpendek, namun boros memori karena harus menyimpan semua simpul dalam satu level sebelum melangkah lebih jauh.



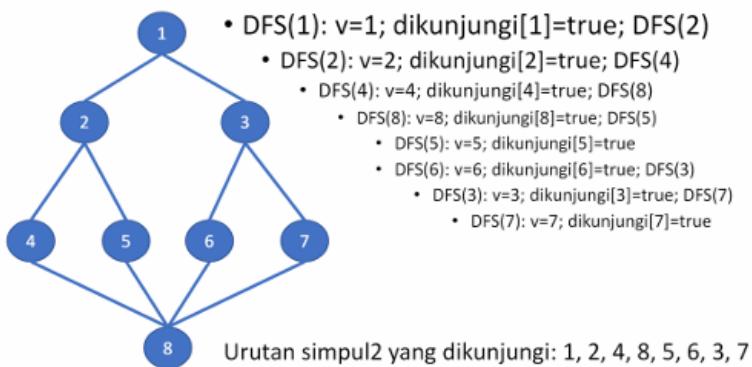
Urutan simpul yang dikunjungi: 1, 2, 3, 4, 5, 6, 7, 8

Gambar 3. Contoh Pencarian pada BFS

#### 1.2. Depth First Search (DFS)

DFS adalah algoritma pencarian yang bekerja dengan prinsip pencarian mendalam, yaitu terus menelusuri satu cabang graf hingga simpul terdalam sebelum kembali (backtrack) ke simpul sebelumnya untuk mencoba jalur lain. DFS menggunakan struktur data tumpukan (stack), baik secara eksplisit maupun melalui rekursi. Meskipun tidak menjamin solusi optimal dan bahkan bisa tersesat ke jalur yang dalam tak berujung, DFS memiliki keunggulan dalam penggunaan memori yang lebih efisien

dibandingkan BFS, dengan kompleksitas ruang  $O(b^m)$  dan waktu  $O(b^m)$ , di mana  $m$  adalah kedalaman maksimum. DFS sangat bermanfaat untuk persoalan yang mengharuskan eksplorasi mendalam, serta menjadi dasar dari teknik backtracking dalam pencarian solusi dengan banyak alternatif seperti pada puzzle, permainan, atau labirin.



Gambar 4. Contoh Pencarian pada DFS

### 1.3. Bidirectional Search

Bidirectional Search adalah algoritma pencarian yang bekerja dengan memulai dua pencarian sekaligus: satu dari simpul awal (*start node*) dan satu dari simpul tujuan (*goal node*), dengan harapan kedua pencarian akan bertemu di tengah. Dengan prinsip ini, ruang pencarian dapat dikurangi secara signifikan karena setiap pencarian hanya perlu menjelajahi hingga setengah kedalaman solusi (misalnya hingga  $d/2$  jika  $d$  adalah kedalaman solusi), sehingga kompleksitas waktu dan ruang menjadi  $O(b^{d/2})$ , jauh lebih efisien dibandingkan pencarian satu arah. Bidirectional Search biasanya menggunakan BFS di kedua arah untuk menjaga optimalitas solusi, namun penerapannya memerlukan syarat bahwa arah pencarian bisa dibalik dan kita dapat memeriksa apakah dua pencarian sudah saling bertemu. Tantangan utama dalam implementasi Bidirectional Search adalah dalam mendeteksi pertemuan simpul dari kedua arah dan merepresentasikan graf secara efisien dari arah tujuan. Algoritma ini sangat efektif pada graf yang besar dan dalam di mana simpul awal dan simpul tujuan sudah diketahui dengan jelas.

## 2. Mengenai Website

Website ini dibangun untuk memenuhi spesifikasi tugas besar 2 IF2211 Strategi Algoritma dengan menekankan algoritma BFS, DFS, dan Bidirectional. Algoritma yang dipilih yakni digunakan untuk mencari resep dari sebuah elemen dalam permainan “Little Alchemy 2” yakni sebuah game yang menampilkan gabungan dari satu elemen dengan elemen lainnya untuk membentuk elemen yang baru. Website ini menyediakan layanan untuk mencari kumpulan resep yang dapat dibuat di dalam permainan “Little Alchemy 2” dengan hanya memasukkan elemen yang ingin dicari lalu memilih mode yang ingin digunakan. Pada website ini juga disediakan visualisasi berupa tree yang bisa dijalankan yang berisikan elemen-elemen yang terlibat dalam resep yang didapatkan. Adapun website ini juga menyediakan fitur multiple recipe yakni pengguna dapat mencari lebih dari satu resep untuk mendapatkan elemen akhirnya. Website ini dibangun menggunakan bahasa Javascript XML untuk frontend, Golang untuk backend, lalu kami menggunakan [Next.js](#) sebagai frameworknya.

## BAB 3: Analisis Pemecahan Masalah

### 1. Langkah-langkah Pemecahan Masalah

Untuk menyelesaikan permasalahan pencarian resep pada permainan Little Alchemy 2, kami mengikuti langkah-langkah sistematis berikut:

#### 1. Pemahaman Masalah

Permainan Little Alchemy 2 mengharuskan pemain menggabungkan elemen dasar (air, earth, fire, water) untuk membentuk elemen turunan hingga mencapai 720 elemen. Tugas ini mensyaratkan pencarian resep menggunakan algoritma Breadth-First Search dan Depth-First Search (DFS), dengan opsi bidirectional search sebagai bonus, serta mendukung pencarian resep terpendek dan banyak resep dengan visualisasi berupa pohon.

#### 2. Pengumpulan Data

Data resep yang dikumpulkan melalui scraping situs Fandom Little Alchemy 2 menggunakan pustaka goquery dalam bahasa Go. Data ini berisi kombinasi elemen (root, left, right, tier) yang disimpan dalam format JSON untuk diolah oleh backend.

#### 3. Pemodelan Masalah

Masalah dimodelkan sebagai graf, dimana setiap elemen adalah simpul (node) dan kombinasi elemen adalah sisi (edge). Struktur data Combination dan Noda pada kode server.go digunakan untuk merepresentasikan hubungan ini.

#### 4. Implementasi Algoritma

Algoritma BFS diimplementasikan dalam fungsi FindRecipeBFS, DFS dalam FindRecipesDFS, dan bidirectional search dalam FindRecipeBidirectional yang menggunakan BFS. Pencarian banyak resep dioptimasi dengan multithreading menggunakan sync dan context dalam FindMultipleRecipes

## 5. Validasi Tier

Setiap kombinasi divalidasi untuk memastikan elemen bahan memiliki tier yang lebih rendah dari elemen hasil, sesuai spesifikasi, menggunakan tierMap dan fungsi isLowerTier.

## 6. Visualisasi dan Pengukuran

Hasil pencarian divisualisasikan sebagai pohon resep, dengan jumlah simpul yang dikunjungi dihitung menggunakan variabel seperti BFSVisitedCount dan DFSVisitedCount. Waktu pencarian juga diukur untuk ditampilkan pada aplikasi.

## 2. Proses Pemetaan Masalah Menjadi Elemen-elemen Algoritma DFS dan BFS

Masalah pencarian resep pada Little Alchemy 2 dipetakan sebagai masalah penelusuran graf, dengan elemen-elemen sebagai simpul dan kombinasi elemen sebagai sisi berarah. Berikut adalah pemetaan dan implementasi algoritma:

### 1. Representasi Graf

Graf direpresentasikan menggunakan struktur data Combination yang menyimpan elemen hasil (Root), element bahan kiri (Left), element bahan kanan (Right), dan tingkat tier element (Tier). Map combinations menyimpan semua kombinasi berdasarkan elemen hasil, sedangkan tierMap menyimpan tingkat tier setiap elemen. Struktur Node digunakan untuk membangun pohon resep, dengan Element sebagai nama elemen dan Left serta Right sebagai subpohon element

### 2. Algoritma BFS

Pada FindRecipeBFS, pencarian dilakukan dari elemen target ke elemen dasar menggunakan *queue*. Algoritma ini menjelajahi semua simpul pada satu tingkat sebelum berpindah ke tingkat berikutnya, memastikan resep terpendek ditemukan. Proses dilakukan dalam dua tahap: tahap pertama mengumpulkan semua elemen yang diperlukan, dan tahap kedua membangun pohon resep dari elemen dasar ke target. Validasi tier dilakukan untuk memastikan bahan memiliki tier lebih rendah.

### 3. Algoritma DFS

Dalam FindRecipeDFS, pencarian dilakukan secara rekursif dengan menelusuri satu cabang hingga kedalaman maksimum sebelum kembali ke cabang lain. Map visited digunakan untuk mencegah siklus, dan validasi tier diterapkan untuk memastikan kombinasi valid. Algoritma ini cocok untuk menemukan satu resep dengan cepat, meskipun tidak menjamin resep terpendek.

### 4. Algoritma Bidirectional (Bonus)

Fungsi FindRecipeBidirectional melakukan pencarian dua arah, dengan BFS maju dari elemen dasar (default Earth) dan BFS mundur dari target, bertemu di simpul perantara. Fungsi pendukung seperti expandForwardBuild dan expandBackwardTrack memastikan ekspansi graf yang efisien.

### 5. Pencarian Multi Recipes

Dalam FindMultipleRecipes, pencarian banyak resep dioptimasi dengan multithreading menggunakan sync.WaitGroup dan sync.Mutex. Fungsi ini membatasi kedalaman pencarian (initialDepth) dan menggunakan recipeCache untuk menyimpan hasil sementara, memastikan resep unik dengan serializeTree.

## 3. Fitur Fungsional dan Arsitektur Aplikasi Web

Fitur Fungsional dari Aplikasi Web adalah sebagai berikut:

- Melakukan pencarian resep dengan Algoritma BFS
- Melakukan pencarian resep dengan Algoritma DFS.
- Melakukan pencarian resep dengan Algoritma Bidirectional.
- Melakukan pencarian multi resep untuk satu elemen.
- Hasil pencarian resep dengan menggunakan graf.
- Menggunakan website Fandom Little Alchemy 2 sebagai sumber data yang digunakan dalam scraping.
- Pengguna dapat memasukkan input elemen, max recipes (untuk multirecipes), dan algoritma pencarian.

Arsitektur aplikasi web kami, terbagi menjadi dua bagian, frontend dan backend. Pengguna awalnya memberi inputan berupa elemen, max recipes (untuk multirecipes), dan algoritma pencarian ke frontend. Frontend kemudian mengirim request ke Backend untuk menyelesaiakannya. Backend melakukan

scraping dari website Fandom Little Alchemy 2 dan menerima data. Backend kemudian menyelesaikan dan mengirim response ke frontend. Frontend kemudian menampilkan hasil tersebut dalam bentuk graf yang dapat dilihat oleh pengguna.

## 4. Contoh Ilustrasi Kasus

Sebagai ilustrasi, kami mengambil kasus pencarian resep untuk elemen Atmosphere yang merupakan element tier 4. Berdasarkan data kombinasi, Atmosphere dapat dibentuk melalui kombinasi valid berikut:

- Air (tier 0) + Planet (tier 3) → valid.
- Air (tier 0) + Sky (tier 5) → tidak valid (tier Sky lebih tinggi dari Atmosphere).

Kemudian fokus pada kombinasi valid Air + Planet. Rantai kombinasi untuk Planet hingga elemen dasar adalah sebagai berikut:

- Planet (tier 3): Continent (tier 2) + Continent (tier 2) → valid.
- Continent (tier 2):
  - Land (tier 1) + Land (tier 1) → valid.
  - Land (tier 1) + Earth (tier 0) → valid.
- Land (tier 1): Earth (tier 0) + Earth (tier 0) → valid.

Element dasar yang terlibat adalah Air dan Earth.

### 1. Pencarian dengan BFS:

Fungsi FindRecipeBFS memulai dari Atmosphere, menelusuri ke Air dan Planet. Karena Air adalah elemen dasar, pencarian berfokus pada Planet, yang mengarah ke Continent, lalu Land, dan akhirnya Earth. Pohon resep yang dihasilkan adalah:

- Atmosphere = Air + Planet
- Planet = Continent + Continent
- Continent = Land + Land
- Land = Earth + Earth

Jumlah simpul yang dikunjungi (diakses melalui GetBFSVisited) diperkirakan sekitar 6 - 10 simpul, karena BFS menjelajahi elemen seperti Atmosphere, Planet, Continent, Land, dan Earth, dengan validasi tier untuk memfilter kombinasi tidak valid (misalnya, Air + Sky)

## 2. Pencarian dengan DFS:

Fungsi FindRecipeDFS menelusuri cabang Air + Planet, langsung ke Air (elemen dasar), lalu ke Planet, Continent, Land, hingga Earth. Karena Planet hanya memiliki satu kombinasi valid dan Land terbatas, DFS menghasilkan pohon serupa dengan BFS. Jumlah simpul yang dikunjungi biasanya lebih banyak daripada BFS, karena DFS menelusuri satu jalur dan berhenti setelah menemukan resep valid.

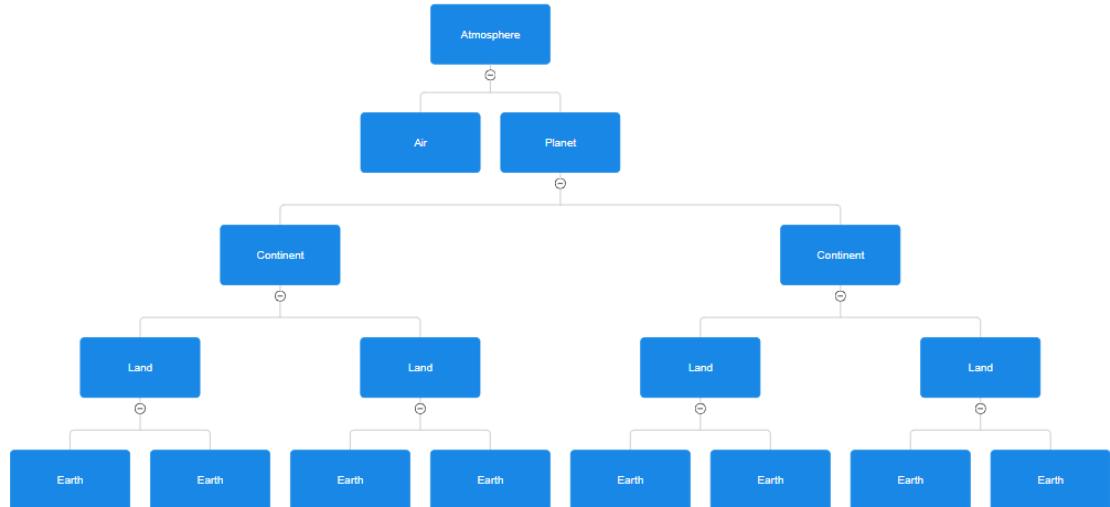
## 3. Pencarian Multi Recipe

Fungsi FindMultipleRecipes dengan maxCount=2 dapat menghasilkan hingga dua resep unik. Karena Atmosphere hanya memiliki satu kombinasi valid (Air + Planet), variasi resep berasal dari Continent, yang memiliki dua kombinasi valid. Contoh dua resep yang mungkin adalah:

- Resep 1:
  - Atmosphere = Air + Planet
  - Planet = Continent + Continent
  - Continent = Land + Land
  - Land = Earth + Earth
- Resep 2:
  - Atmosphere = Air + Planet
  - Planet = Continent + Continent
  - Continent = Land + Earth
  - Land = Earth + Earth

Pohon resep divisualisasikan dengan simpul daun sebagai elemen dasar (Air, Earth), sesuai spesifikasi Gambar 3. Jumlah simpul yang dikunjungi (diakses melalui GetMultiVisited) lebih tinggi, sekitar 10-15 simpul, karena pencarian menjelajahi beberapa jalur untuk menemukan resep unik, dengan optimasi multithreading dan caching.

Berikut adalah visualisasi pohon resep untuk Resep 1:



*Gambar 5. Gambar Tree Recipe Atmosphere*

Pohon ini menunjukkan bahwa Atmosphere dibentuk dari Air dan Planet, dengan Planet memerlukan dua Continent, masing-masing dibentuk dari dua Land, dan setiap Land dari dua Earth. Validasi tier memastikan hanya kombinasi dengan tier lebih rendah yang digunakan, seperti yang diimplementasikan dalam fungsi `isLowerTier`.

## BAB 4: Implementasi dan Pengujian

### 4.1 Spesifikasi Teknis Program

Bagian ini menjelaskan spesifikasi teknis program yang dibangun, mencakup struktur data serta fungsi dan prosedur yang diimplementasikan dalam berkas server.go pada backend aplikasi. Implementasi menggunakan bahasa Golang untuk mendukung pemrosesan data resep dan pencarian menggunakan algoritma BFS, DFS, dan bidirectional search.

#### 4.1.1 Struktur Data

Program menggunakan beberapa struktur data utama untuk merepresentasikan elemen, kombinasi, dan pohon resep dalam permainan Little Alchemy 2. Struktur data ini dirancang untuk mendukung pencarian resep.

##### 1. Combination

Struktur ini merepresentasikan satu kombinasi untuk membentuk elemen baru. Struktur ini didefinisikan sebagai berikut:

```
type Combination struct {
    Root  string `json:"root"`
    Left   string `json:"left"`
    Right  string `json:"right"`
    Tier   int     `json:"tier,string"`
}
```

Atribut Root adalah elemen hasil, Left dan Right adalah elemen bahan, dan Tier adalah tingkatan elemen berdasarkan kompleksitasnya.

##### 2. Node

Struktur ini merepresentasikan simpul dalam pohon resep, digunakan untuk membangun visualisasi resep.

```
type Node struct {
    Element string
    Left     *Node
    Right    *Node
}
```

Atribut Element menyimpan nama elemen, sedangkan Left dan Right adalah pointer ke subpohon bahan pembentuk.

### 3. RecipePath

Struktur ini digunakan untuk menyimpan jalur bahan dalam pencarian bidirectional.

```
type RecipePath struct {
    Left  string
    Right string
}
```

Struktur ini menyimpan pasangan elemen bahan (Left dan Right) untuk melacak jalur dalam pencarian dua arah.

### 4. Map Kombinasi dan Tier

- combinations: Map dengan kunci berupa nama elemen dan nilai berupa daftar Combination yang menghasilkan elemen tersebut.
- tierMap: Map yang menyimpan tingkatan tier setiap elemen.
- reverseMap: Map untuk pencarian bidirectional, menyimpan hubungan terbalik dari kombinasi (elemen bahan ke elemen hasil).

### 5. Counter

Variabel global seperti BFSVisitedCount, DFSVisitedCount, MultiVisitedCount (dengan atomic untuk thread-safety), dan BidirectionalVisitedCount di gunakan untuk menghitung jumlah simpul yang dikunjungi selama pencarian.

#### 4.1.2 Fungsi dan Prosedur

Program mengimplementasikan sejumlah fungsi utama untuk mendukung pencarian resep dan pengelolaan data. Berikut adalah sejumlah fungsi utama untuk mendukung pencarian resep dan pengelolaan data. Berikut fungsi-fungsi utama beserta deskripsinya:

##### 1. LoadCombinations(filename string) error

```
func LoadCombinations(filename string) error {
    data, err := os.ReadFile(filename)
    if err != nil {
        return err
    }
    var combinations []Combination
    for _, line := range strings.Split(string(data), "\n") {
        combination := new(Combination)
        err = combination.UnmarshalText([]byte(line))
        if err != nil {
            return err
        }
        combinations = append(combinations, *combination)
    }
    return nil
}
```

```

    }

    var raw []Combination
    if err := json.Unmarshal(data, &raw); err != nil
    {
        return err
    }
    combinations = make(map[string][]Combination)
    tierMap = make(map[string]int)
    reverseMap = make(map[string][]string)

    for _, c := range raw {
        combinations[c.Root] =
            append(combinations[c.Root], c)
        tierMap[c.Root] = c.Tier
        reverseMap[c.Left] =
            append(reverseMap[c.Left], c.Root)
        reverseMap[c.Right] =
            append(reverseMap[c.Right], c.Root)
    }
    return nil
}

```

- Fungsi: Memuat data kombinasi dari berkas JSON ke dalam combinations, tierMap, dan reverseMap.
- Input: Nama berkas JSON.
- Output: error jika terjadi kesalahan, nil jika berhasil.
- Deskripsi: Membaca berkas JSON, melakukan unmarshal ke slice Combination, dan mengisi map global untuk pencarian.

## 2. isBasic(element string) bool

```

func isBasic(element string) bool {
    basics := map[string]bool{
        "Earth": true,
        "Air": true,
        "Water": true,
        "Fire": true,
        "Time": true,
    }
    return basics[element]
}

```

- Fungsi: Memeriksa apakah elemen adalah elemen dasar (Earth, Air, Water, Fire, Time).
- Input: Nama elemen.
- Output: true jika elemen dasar, false jika tidak.

### 3. FindRecipeBFS(target string) \*Node

```
func FindRecipeBFS(target string) *Node {
    fmt.Printf("\n==== Starting BFS search for: %s\n",
    target)

    if isBasic(target) {
        fmt.Printf("Found basic element: %s\n",
target)
        BFSVisitedCount = 1
        return &Node{Element: target}
    }

    if _, exists := combinations[target]; !exists {
        fmt.Printf("Element %s not found in
combinations\n", target)
        BFSVisitedCount = 0
        return nil
    }

    fmt.Printf("Found %d combinations for %s\n",
len(combinations[target]), target)
    visited := make(map[string]bool)
    recipeMap := make(map[string]*Node)
    queue := []string{target}
    BFSVisitedCount = 0

    // First pass: collect all possible combinations
    fmt.Println("\nFirst pass: Collecting
combinations...")
    for len(queue) > 0 {
        current := queue[0]
        queue = queue[1:]
        if visited[current] {
            continue
        }
    }
}
```

```

        visited[current] = true
        BFSVisitedCount++
        fmt.Printf("Visiting: %s (visited count:
%d)\n", current, BFSVisitedCount)

        if isBasic(current) {
            fmt.Printf("Found basic element:
%s\n", current)
            recipeMap[current] = &Node{Element:
current}
            continue
        }

        // Add all possible combinations to the
queue
        for _, comb := range combinations[current]
{
            // Check if both ingredients are of
lower tier than the target
            if tierMap[comb.Left] <
tierMap[current] && tierMap[comb.Right] <
tierMap[current] {
                fmt.Printf(" Checking
combination: %s (tier %d) + %s (tier %d) = %s (tier
%d)\n",
                    comb.Left,
                    tierMap[comb.Left], comb.Right, tierMap[comb.Right],
                    comb.Root, comb.Tier)
                if !visited[comb.Left] {
                    queue = append(queue,
                    comb.Left)
                    fmt.Printf("     Added to
queue: %s\n", comb.Left)
                }
                if !visited[comb.Right] {
                    queue = append(queue,
                    comb.Right)
                    fmt.Printf("     Added to
queue: %s\n", comb.Right)
                }
            } else {
                fmt.Printf("     Skipping invalid
combination: %s (tier %d) + %s (tier %d) = %s (tier
%d)\n",
                    comb.Left,
                    tierMap[comb.Left], comb.Right, tierMap[comb.Right],

```

```

        comb.Root, comb.Tier)
    }
}
}

// Second pass: build recipes from basic
elements up
fmt.Println("\nSecond pass: Building
recipes...")
changed := true
for changed {
    changed = false
    for elem := range visited {
        if recipeMap[elem] != nil {
            continue
        }

        // Try all combinations for this
element
        for _, comb := range
combinations[elem] {
            // Check if both ingredients are
of lower tier than the target
            if tierMap[comb.Left] <
tierMap[elem] && tierMap[comb.Right] < tierMap[elem]
{
                leftRecipe :=
recipeMap[comb.Left]
                rightRecipe :=
recipeMap[comb.Right]
                if leftRecipe != nil &&
rightRecipe != nil {
                    fmt.Printf("Found
recipe for %s: %s + %s\n",
elem, comb.Left,
comb.Right)
                    recipeMap[elem] =
&Node{
                        Element: elem,
                        Left:
leftRecipe,
                        Right:
rightRecipe,
                    }
                    changed = true
                    break
                }
            }
        }
    }
}

```

```

        }
    }
}

result := recipeMap[target]
if result != nil {
    fmt.Printf("\nSuccessfully found recipe for
%s\n", target)
} else {
    fmt.Printf("\nNo valid recipe found for
%s\n", target)
}
return result
}

```

- Fungsi: Mencari satu resep terpendek untuk elemen target menggunakan BFS.
- Input: Nama elemen target.
- Output: Pointer ke Node yang merepresentasikan pohon resep, atau nil jika tidak ditemukan.
- Deskripsi: Menggunakan antrian untuk menelusuri elemen dari target ke elemen dasar, memastikan tier bahan lebih rendah, dan membangun pohon resep dari bawah ke atas.

#### 4. FindRecipeDFS(target string, visited map[string]bool) \*Node

```

func FindRecipeDFS(target string, visited
map[string]bool) *Node {
    if _, exists := combinations[target]; !exists &&
!isBasic(target) {
        return nil
    }

    if isBasic(target) {
        DFSVisitedCount++
        return &Node{Element: target}
    }

    if visited == nil {
        visited = make(map[string]bool)
    }
}
```

```

        DFSVisitedCount = 0
    }

    if visited[target] {
        return nil
    }

    visited[target] = true
    DFSVisitedCount++
    defer func() { visited[target] = false }()

    for _, comb := range combinations[target] {
        if tierMap[comb.Left] < tierMap[target] &&
tierMap[comb.Right] < tierMap[target] {
            left := FindRecipeDFS(comb.Left,
visited)
            if left == nil {
                continue
            }
            right := FindRecipeDFS(comb.Right,
visited)
            if right != nil {
                return &Node{Element: target,
Left: left, Right: right}
            }
        }
    }

    return nil
}

```

- Fungsi: Mencari satu resep untuk elemen target menggunakan DFS.
- Input: Nama elemen target dan peta visited untuk mencegah siklus.
- Output: Pointer ke Node pohon resep, atau nil jika tidak ditemukan.
- Deskripsi: Melakukan pencarian rekursif, menelusuri satu cabang hingga elemen dasar, dengan validasi tier.

## 5. FindRecipeBidirectional(target string) \*Node

```

func FindRecipeBidirectional(target string,
startElement string) *Node {

```

```

        fmt.Printf("\n== Starting Bidirectional Search\n")
        fmt.Printf("Target: %s (Tier: %d)\n", target,
tierMap[target])
        fmt.Printf("Start Element: %s (Tier: %d)\n",
startElement, tierMap[startElement])

        if isBasic(target) {
            fmt.Printf("Target is a basic element,
returning direct node\n")
            return &Node{Element: target}
        }
        if _, exists := combinations[target]; !exists {
            fmt.Printf("Target element not found in
combinations\n")
            return nil
        }
        if !isBasic(startElement) {
            fmt.Printf("Start element is not a basic
element\n")
            return nil
        }

        // Forward search (start → target)
        forwardVisited := make(map[string]*Node)
        forwardQueue := []string{startElement}
        forwardVisited[startElement] = &Node{Element:
startElement}

        // Backward search (target → basic)
        backwardVisited := make(map[string]bool)
        backwardQueue := []string{target}
        backwardVisited[target] = true

        BidirectionalVisitedCount = 2 // Start with both
elements
        fmt.Printf("Initialized bidirectional search\n")

        // Main search loop
        for len(forwardQueue) > 0 && len(backwardQueue) >
0 {
            // Forward expansion
            currentForward := forwardQueue[0]
            forwardQueue = forwardQueue[1:]
            fmt.Printf("\nForward exploring from: %s
(Tier: %d)\n", currentForward,

```



```

append(forwardQueue, c.Root)
    BidirectionalVisitedCount++
}
}
}

// Backward expansion
currentBackward := backwardQueue[0]
backwardQueue = backwardQueue[1:]
fmt.Printf("\nBackward exploring from: %s
(Tier: %d)\n", currentBackward,
tierMap[currentBackward])

// Check if current backward node is in
forward visited
if node, exists :=
forwardVisited[currentBackward]; exists {
    fmt.Printf("Found intersection at: %s\n",
currentBackward)
    // Build path from start to intersection
    forwardPath := node
    // Build path from intersection to target
    backwardPath :=
buildBackwardPath(currentBackward, target,
backwardVisited)
    if backwardPath != nil {
        return &Node{
            Element: target,
            Left:    forwardPath,
            Right:   backwardPath,
        }
    }
}

// Expand backward
for _, comb := range
combinations[currentBackward] {
    for _, ingredient := range
[]string{comb.Left, comb.Right} {
        if !backwardVisited[ingredient] &&
tierMap[ingredient] < tierMap[currentBackward] {
            fmt.Printf(" Backward found
ingredient: %s\n", ingredient)
            backwardVisited[ingredient] =
true
        }
    }
}

```

```

        backwardQueue =
append(backwardQueue, ingredient)
    BidirectionalVisitedCount++
}
}
}

fmt.Printf("\nNo recipe found from %s to %s\n",
startElement, target)
return nil
}

```

- Fungsi: Mencari resep menggunakan pencarian bidirectional (bonus).
- Input: Nama elemen target.
- Output: Pointer ke Node pohon resep, atau nil jika tidak ditemukan.
- Deskripsi: Melakukan BFS dari elemen dasar ke target menggunakan reverseMap, mengurangi jumlah simpul yang dikunjungi.

## 6. FindMultipleRecipes(target string, maxCount int) []\*Node

```

func FindMultipleRecipes(target string, maxCount int,
algorithm string) []*Node {
    if isBasic(target) {
        atomic.StoreInt32(&MultiVisitedCount, 1)
        return []*Node{&Element: target}
    }
    if _, exists := combinations[target]; !exists {
        return nil
    }

    atomic.StoreInt32(&MultiVisitedCount, 0)
    var results []*Node
    resultChan := make(chan *Node, maxCount*16) // Increased buffer size
    var wg sync.WaitGroup
    semaphore := make(chan struct{}, 300) // Increased concurrent searches
    var seen sync.Map
    validCombos := []Combination{

```

```

        targetTier := tierMap[target]

        // Filter and sort combinations by tier
        difference and complexity
        for _, c := range combinations[target] {
            if tierMap[c.Left] < targetTier &&
tierMap[c.Right] < targetTier {
                // Calculate tier difference to
                prioritize combinations with ingredients closer to
                target tier
                leftTierDiff := targetTier -
tierMap[c.Left]
                rightTierDiff := targetTier -
tierMap[c.Right]
                avgTierDiff := (leftTierDiff +
rightTierDiff) / 2

                // Add tier difference information to
                the combination
                c.Tier = avgTierDiff
                validCombos = append(validCombos, c)
            }
        }

        // Sort combinations by tier difference and
        complexity
        sort.SliceStable(validCombos, func(i, j int)
bool {
            // First sort by tier difference (prefer
            combinations with ingredients closer to target tier)
            if validCombos[i].Tier !=
validCombos[j].Tier {
                return validCombos[i].Tier <
validCombos[j].Tier
            }
            // Then sort by complexity
            return
len(validCombos[i].Left)+len(validCombos[i].Right) <
len(validCombos[j].Left)+len(validCombos[j].Right)
        })

        // Create a context with timeout
        ctx, cancel :=
context.WithTimeout(context.Background(),
180*time.Second) // Increased timeout
    
```

```

    defer cancel()

    // Function to explore a single combination
    exploreCombination := func(c Combination) {
        defer wg.Done()
        defer func() { <-semaphore }()
        variations := []struct {
            left string
            right string
        } {
            {c.Left, c.Right},
            {c.Right, c.Left},
        }

        for _, v := range variations {
            // Try different combinations for both
            left and right sides
            for _, leftComb := range
            combinations[target] {
                if tierMap[leftComb.Left] <
                targetTier && tierMap[leftComb.Right] < targetTier {
                    visited :=
                    make(map[string]bool)
                    var left *Node
                    // Use the specified
                    algorithm to find the left ingredient
                    switch algorithm {
                    case "bfs":
                        left =
                        FindRecipeBFS(v.left)
                    atomic.AddInt32(&MultiVisitedCount,
                    int32(GetBFSSvisited())))
                    case "dfs":
                        left =
                        FindRecipeDFS(v.left, visited)
                    atomic.AddInt32(&MultiVisitedCount,
                    int32(GetDFSSvisited())))
                    case "bidirectional":
                        // For bidirectional,
                        we'll use the first basic element as start
                        left =

```

```

FindRecipeBidirectional(v.left, "Earth")

atomic.AddInt32(&MultiVisitedCount,
int32(GetBidirectionalVisited()))
    default:
        left =
exploreRecipe(v.left, visited, &MultiVisitedCount,
algorithm)
    }

        if left == nil {
            continue
        }

            for _, rightComb := range
combinations[target] {
                if
tierMap[rightComb.Left] < targetTier &&
tierMap[rightComb.Right] < targetTier {
                    rightVisited :=
copyVisitedMap(visited)
                    var right *Node

                        // Use the
specified algorithm to find the right ingredient
                        switch algorithm {
                            case "bfs":
                                right =
FindRecipeBFS(v.right)

atomic.AddInt32(&MultiVisitedCount,
int32(GetBFSSVisited()))
                            case "dfs":
                                right =
FindRecipeDFS(v.right, rightVisited)

atomic.AddInt32(&MultiVisitedCount,
int32(GetDFSSVisited()))
                            case
"bidirectional":
                                // For
bidirectional, we'll use the first basic element as
start
                                right =
FindRecipeBidirectional(v.right, "Earth")

```



```

        select {
        case <-ctx.Done():
            break
        default:
            wg.Add(1)
            semaphore <- struct{}{}
            go exploreCombination(comb)
        }
    }

    // Close result channel when all goroutines are
done
    go func() {
        wg.Wait()
        close(resultChan)
    }()
}

// Collect results with respect to maxCount
results = make([]*Node, 0, maxCount)
for recipe := range resultChan {
    results = append(results, recipe)
    if len(results) >= maxCount {
        cancel()
        break
    }
}

// If we found more recipes than requested,
randomly select maxCount recipes
if len(results) > maxCount {
    rand.Seed(time.Now().UnixNano())
    rand.Shuffle(len(results), func(i, j int) {
        results[i], results[j] = results[j],
results[i]
    })
    results = results[:maxCount]
}

return results
}

```

- Fungsi: Mencari hingga maxCount resep unik untuk elemen target dengan multithreading.
- Input: Nama elemen target dan jumlah maksimum resep.

- Output: Slice dari pointer Node yang merepresentasikan pohon-pohon resep.
- Deskripsi: Menggunakan multithreading dengan sync.WaitGroup dan sync.Mutex, membatasi kedalaman pencarian, dan menyimpan resep unik menggunakan serializeTree.

## 7. FindRecipeBidirectional(target string, startElement string) \*Node:

```
func FindRecipeBidirectional(target string,
startElement string) *Node {
    fmt.Printf("\n==== Starting Bidirectional Search
====\n")
    fmt.Printf("Target: %s (Tier: %d)\n", target,
tierMap[target])
    fmt.Printf("Start Element: %s (Tier: %d)\n",
startElement, tierMap[startElement])

    if isBasic(target) {
        fmt.Printf("Target is a basic element,
returning direct node\n")
        return &Node{Element: target}
    }
    if _, exists := combinations[target]; !exists {
        fmt.Printf("Target element not found in
combinations\n")
        return nil
    }
    if !isBasic(startElement) {
        fmt.Printf("Start element is not a basic
element\n")
        return nil
    }

    // Forward search (start → target)
    forwardVisited := make(map[string]*Node)
    forwardQueue := []string{startElement}
    forwardVisited[startElement] = &Node{Element:
startElement}

    // Backward search (target → basic)
    backwardVisited := make(map[string]bool)
    backwardQueue := []string{target}
    backwardVisited[target] = true
```

```

        BidirectionalVisitedCount = 2 // Start with both
elements
        fmt.Printf("Initialized bidirectional search\n")

        // Main search loop
        for len(forwardQueue) > 0 && len(backwardQueue) >
0 {
            // Forward expansion
            currentForward := forwardQueue[0]
            forwardQueue = forwardQueue[1:]
            fmt.Printf("\nForward exploring from: %s
(Tier: %d)\n", currentForward,
tierMap[currentForward])

            // Check if current forward node is in
backward visited
            if backwardVisited[currentForward] {
                fmt.Printf("Found intersection at: %s\n",
currentForward)
                // Build path from start to intersection
                forwardPath :=
forwardVisited[currentForward]
                // Build path from intersection to target
                backwardPath :=
buildBackwardPath(currentForward, target,
backwardVisited)
                if backwardPath != nil {
                    return &Node{
                        Element: target,
                        Left:    forwardPath,
                        Right:   backwardPath,
                    }
                }
            }

            // Expand forward
            for _, comb := range combinations {
                for _, c := range comb {
                    if (c.Left == currentForward ||

c.Right == currentForward) &&
forwardVisited[c.Left] != nil &&
forwardVisited[c.Right] != nil &&
tierMap[c.Left] < tierMap[c.Root] &&
tierMap[c.Right] < tierMap[c.Root] {

                        if _, exists :=
```

```

forwardVisited[c.Root]; !exists {
    fmt.Printf("  Forward found:
%s + %s = %s\n", c.Left, c.Right, c.Root)
    forwardVisited[c.Root] =
&Node{
    Element: c.Root,
    Left: nil,
    Right: nil}
}

forwardVisited[c.Left],
forwardVisited[c.Right],
forwardQueue =
append(forwardQueue, c.Root)
BidirectionalVisitedCount++
}

}

// Backward expansion
currentBackward := backwardQueue[0]
backwardQueue = backwardQueue[1:]
fmt.Printf("\nBackward exploring from: %s
(Tier: %d)\n", currentBackward,
tierMap[currentBackward])

// Check if current backward node is in
forward visited
if node, exists :=
forwardVisited[currentBackward]; exists {
    fmt.Printf("Found intersection at: %s\n",
currentBackward)
    // Build path from start to intersection
    forwardPath := node
    // Build path from intersection to target
    backwardPath :=
buildBackwardPath(currentBackward, target,
backwardVisited)
    if backwardPath != nil {
        return &Node{
            Element: target,
            Left: forwardPath,
            Right: backwardPath,
        }
    }
}

```

```

        // Expand backward
        for _, comb := range
combinations[currentBackward] {
            for _, ingredient := range
[]string{comb.Left, comb.Right} {
                if !backwardVisited[ingredient] &&
tierMap[ingredient] < tierMap[currentBackward] {
                    fmt.Printf(" Backward found
ingredient: %s\n", ingredient)
                    backwardVisited[ingredient] =
true
                    backwardQueue =
append(backwardQueue, ingredient)
                    BidirectionalVisitedCount++
                }
            }
        }

        fmt.Printf("\nNo recipe found from %s to %s\n",
startElement, target)
        return nil
    }
}

```

- Fungsi: Mencari resep menggunakan bidirectional search.
- Input: Nama elemen target dan elemen dasar awal (default Earth).
- Output: Pointer ke Node pohon resep, atau nil jika tidak ditemukan.
- Deskripsi: Melakukan BFS maju dari startElement dan BFS mundur dari target, bertemu di simpul perantara. Fungsi expandForwardBuild dan expand BackwardTrack mendukung ekspansi graf, sementara buildBackwardPath mem bangun jalur mundur. Berikut adalah cuplikan kode utama:

#### 8. Fungsi Pembantu:

- treeDepth(node \*Node) int: Menghitung kedalaman maksimum pohon resep.
- serializeTree(n \*Node) string: Mengubah pohon menjadi string unik untuk mendeteksi duplikat.
- IsBasic(element string) bool: Versi eksternal dari isBasic untuk akses frontend.

- GetCombinations(element string) []Combination: Mengembalikan daftarkom binasi untuk elemen tertentu.
- isLowerTier(c Combination) bool: Memeriksa apakah kombinasi memiliki bahan dengan tier lebih rendah.
- GetBFSVisited(), GetDFSVisited(), GetMultiVisited(), GetBidirectionalVisited(): Mengembalikan jumlah simpul yang dikunjungi untuk masing-masing metode pencarian.
- expandForwardBuild(queue []string, visited map[string]\*Node, otherVisited map[string]bool, intersection \*string) ([]string, bool): Mendukung ekspansi BFS maju dalam bidirectional search.
- expandBackwardTrack(queue []string, visited map[string]bool, otherVisited map[string]\*Node, intersection \*string) ([]string, bool): Mendukung ekspansi BFS mundur dalam bidirectional search.
- buildBackwardPath(from, to string, visited map[string]bool) \*Node: Membangun jalur mundur dari simpul perantara ke target.
- handleSearch(w http.ResponseWriter, r \*http.Request): Menangani permintaan HTTP untuk pencarian resep, mendukung mode single dan multiple.
- convertRecipeToPath(node \*Node) []Step: Mengonversi pohon resep menjadi daftar langkah untuk respons JSON.

## 4.2 Penjelasan Tata Cara Penggunaan Program

### 4.2.1 Cara Menjalankan Program secara Lokal

Untuk menjalankan program, diperlukan dua terminal .Terminal ke-1 akan menjelaskan bagian frontend dan terminal ke-2 akan menjalankan bagian backend. Aplikasi hanya akan dapat berjalan jika kedua bagian berjalan pada terminal yang berbeda. Langkah-langkah untuk menjalankan kedua program adalah sebagai berikut:

1. Clone repository: [https://github.com/ivant8k/Tubes2\\_SOS](https://github.com/ivant8k/Tubes2_SOS)
2. Buka Terminal dan ketik `cd src`
3. Untuk backend, masuk ke folder backend dengan `cd backend` kemudian ketik `go run server.go`
4. Untuk frontend, masuk ke folder frontend dengan `cd frontend` kemudian ketik `npm install` dan `npm run dev`.

Setelah kedua bagian, frontend dan backend, sudah dijalankan maka website siap digunakan. Untuk membuka website, buka tautan <http://localhost:3000/> pada browser.

## 4.2.2 Cara Menjalankan Program secara Online

Kami sudah melakukan *deployment* pada proyek ini menggunakan vercel dan railway dan bisa digunakan dengan mengakses link berikut: <https://alchemix.vercel.app/>

## 4.2.1 Cara Menjalankan Program dengan Docker

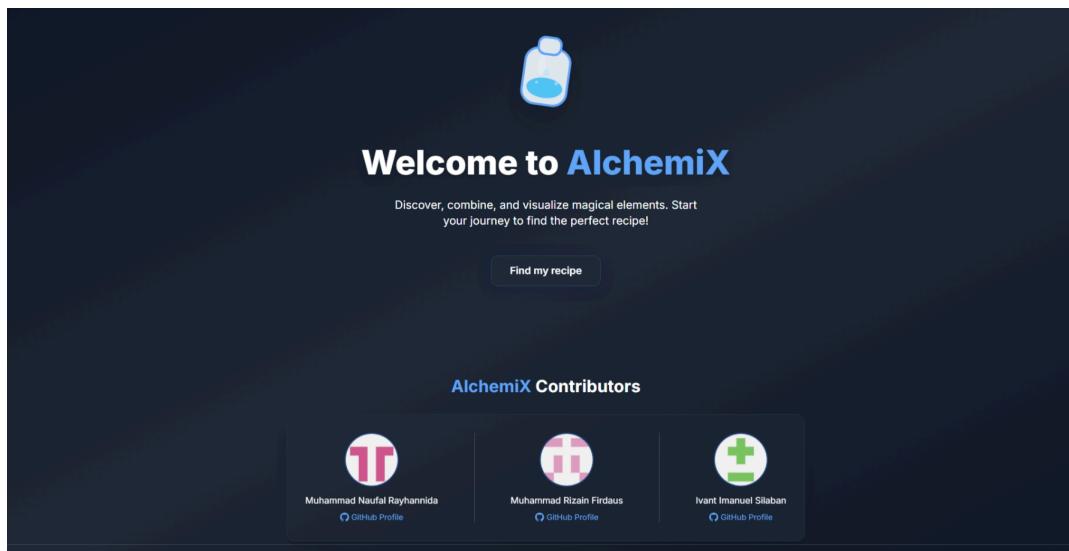
Aplikasi ini telah di-containerize menggunakan Docker untuk memudahkan proses instalasi dan deployment. Berikut adalah langkah-langkah untuk menjalankan aplikasi menggunakan Docker:

1. Pastikan Docker dan Docker Compose terinstal di sistem anda.
2. Clone Repository: [https://github.com/ivant8k/Tubes2\\_SOS](https://github.com/ivant8k/Tubes2_SOS)
3. Jalankan Aplikasi dengan Docker Compose: **docker compose up --build**  
Jika anda sudah membangun image sebelumnya, cukup gunakan: **docker compose up**
4. Akses Aplikasi: <http://localhost:3000>
5. Hentikan Aplikasi Jika sudah selesai digunakan dengan **Ctrl + C** di tempat **docker compose up** dijalankan. Dan untuk menghapus container, jalankan: **docker compose down**

## 4.2.4 Cara Penggunaan Program

### 4.2.4.1 Tampilan Awal

Berikut tampilan awal program ketika dibuka:

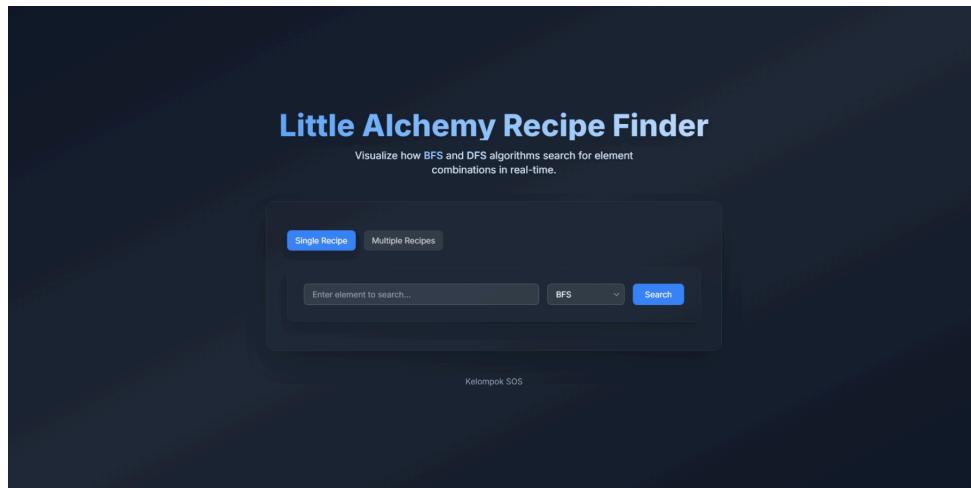


Gambar 6. Tampilan Awal Program

Setelah program dibuka, silahkan klik tombol **Find my recipe** untuk memulai menjalankan program.

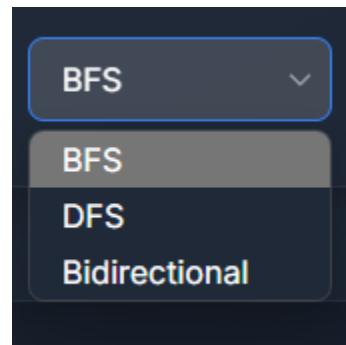
#### 4.2.4.2 Tampilan Single Recipe

Berikut adalah tampilan awal mode single recipe.



Gambar 7. Tampilan Awal Mode Single Recipe

Berikut adalah pilihan algoritma untuk mode single recipe



Gambar 8. Tampilan Pilihan Algoritma

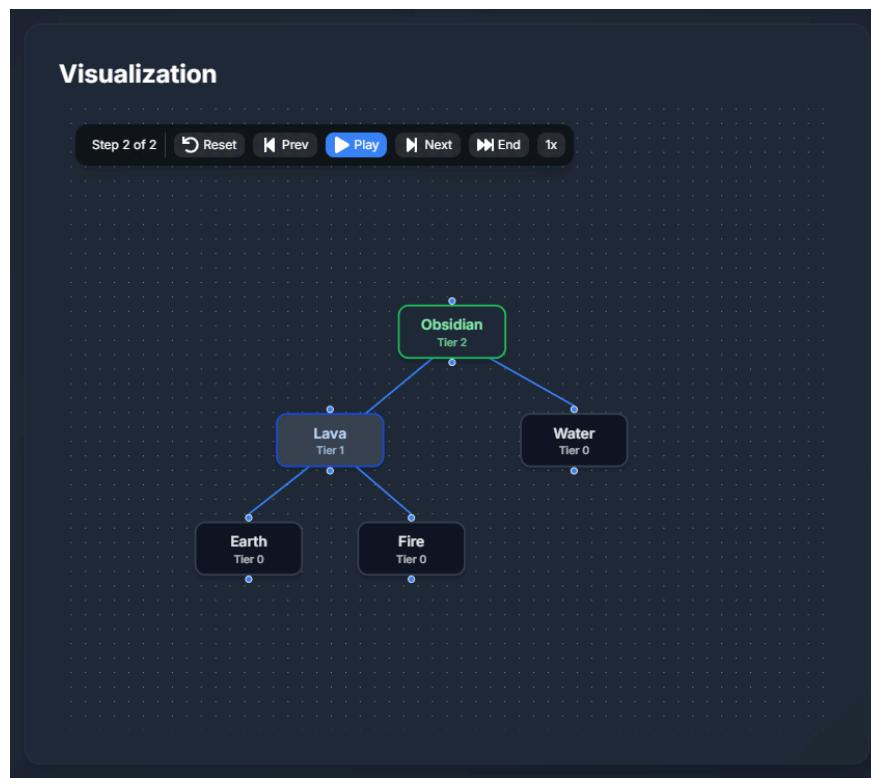
Berikut adalah contoh output ketika memasukkan Obsidian sebagai elemen yang ingin dicari menggunakan algoritma BFS

The screenshot shows a search interface with the following details:

- Search Query: Obsidian
- Search Algorithm: BFS
- Search Results: Element found after visiting 5 nodes
- Time: 0.04 ms
- Target: Obsidian [Tier 2]
- Path:
  1. Earth [Tier 0] + Fire [Tier 0] = Lava [Tier 1]
  2. Lava [Tier 1] + Water [Tier 0] = Obsidian [Tier 2]

Gambar 9. Tampilan Hasil

Berikut adalah visualisasi dari resep 1..



Gambar 10. Tampilan Visualisasi

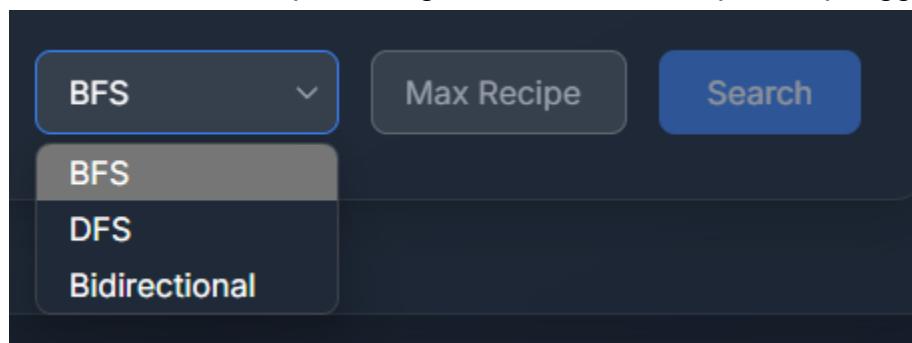
#### 4.2.4.3 Tampilan Multi Recipe

Berikut adalah tampilan awal mode multi recipe.



Gambar 11. Tampilan Awal Mode Multi Recipe

Berikut adalah pilihan algoritma dan max recipe dari pengguna



Gambar 12. Tampilan Input dari Pengguna

Berikut adalah contoh output recipe 1 dengan elemen yang dicari adalah Glass, dengan metode DFS dan max recipe adalah 3

The screenshot shows a search interface with the following elements:

- Top navigation: "Single Recipe" (grayed out) and "Multiple Recipes" (highlighted blue).
- Search bar: "Glass", "DFS", "3", and a "Search" button.
- Section title: "Search Results".
- Text: "Element found after visiting 1661 nodes" and "0.34 ms".
- Text: "Target: Glass Tier 4".
- Buttons: "Recipe 1" (highlighted blue) and "Recipe 2".
- Section title: "Recipe 1 Path:". This section lists six steps:
  - Air Tier 0 + Air Tier 0 = Pressure Tier 1
  - Earth Tier 0 + Pressure Tier 1 = Stone Tier 2
  - Stone Tier 2 + Air Tier 0 = Sand Tier 3
  - Fire Tier 0 + Fire Tier 0 = Energy Tier 1
  - Air Tier 0 + Energy Tier 1 = Heat Tier 2
  - Sand Tier 3 + Heat Tier 2 = Glass Tier 4

Gambar 13. Contoh Hasil Recipe 1

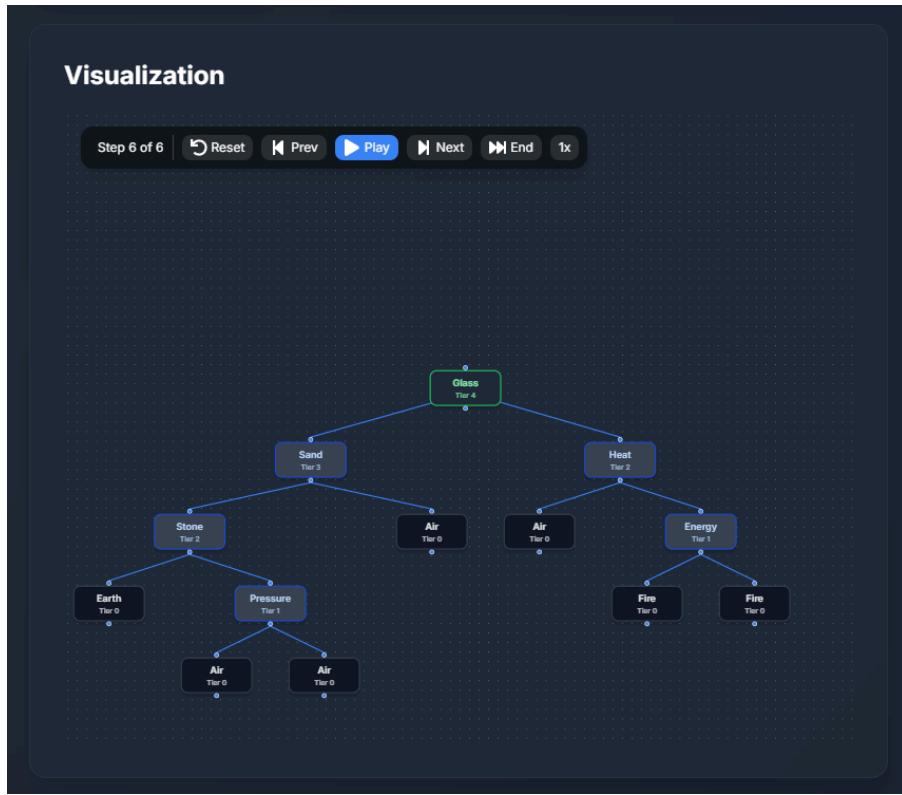
Berikut adalah contoh output recipe 2 dengan elemen yang dicari adalah Glass, dengan metode DFS dan max recipe adalah 3.

The screenshot shows a search interface with the following elements:

- Top navigation: "Single Recipe" (grayed out) and "Multiple Recipes" (highlighted blue).
- Search bar: "Glass", "DFS", "3", and a "Search" button.
- Section title: "Search Results".
- Text: "Element found after visiting 1661 nodes" and "0.34 ms".
- Text: "Target: Glass Tier 4".
- Buttons: "Recipe 1" and "Recipe 2" (highlighted blue).
- Section title: "Recipe 2 Path:". This section lists four steps:
  - Air Tier 0 + Air Tier 0 = Pressure Tier 1
  - Earth Tier 0 + Pressure Tier 1 = Stone Tier 2
  - Stone Tier 2 + Air Tier 0 = Sand Tier 3
  - Sand Tier 3 + Fire Tier 0 = Glass Tier 4

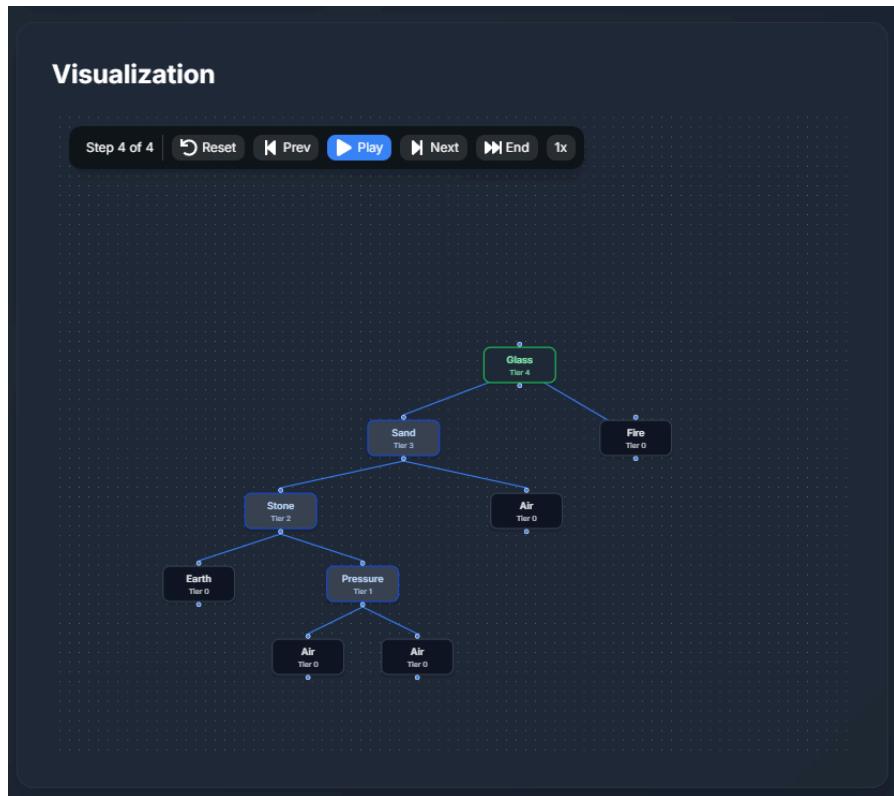
*Gambar 14. Contoh Hasil recipe 2*

Berikut adalah visualisasi untuk resep 1.



*Gambar 15. Contoh visualisasi recipe 1*

Berikut adalah visualisasi untuk resep 2



Gambar 16. Contoh Hasil Recipe 2

## 4.3 Hasil Pengujian

### 4.3.1 Hasil Pengujian untuk Single Recipe

1. Test Case: Obsidian
  - a. BFS

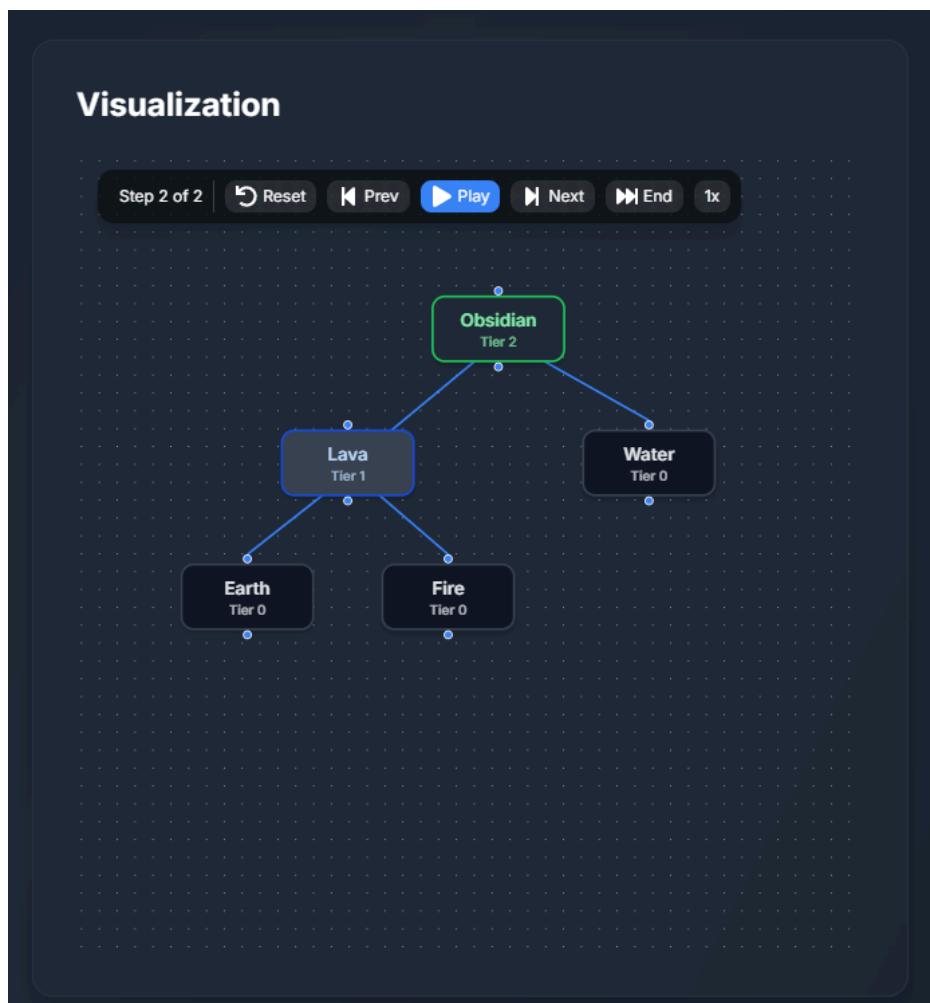
## Search Results

Element found after visiting 5 nodes 0.04 ms

Target: Obsidian Tier 2

**Path:**

1. Earth Tier 0 + Fire Tier 0 = Lava Tier 1
2. Lava Tier 1 + Water Tier 0 = Obsidian Tier 2



b. DFS

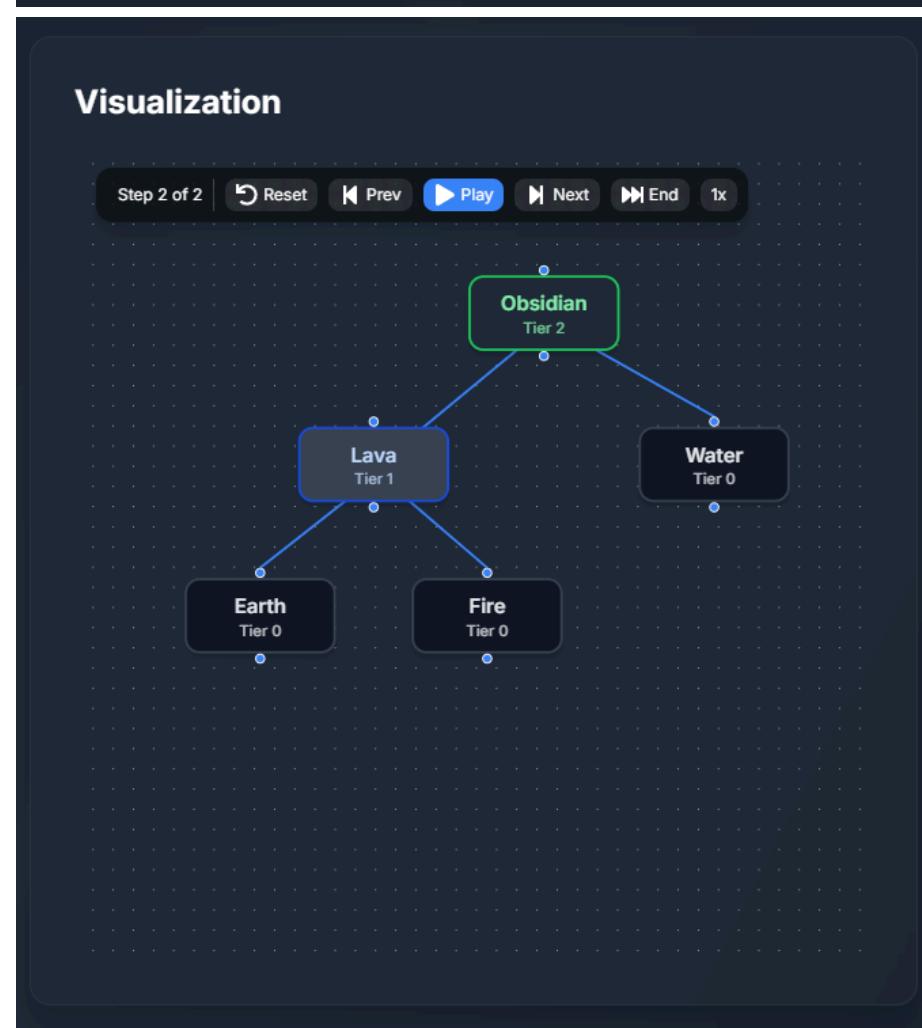
## Search Results

Element found after visiting 5 nodes 0.02 ms

Target: Obsidian Tier 2

Path:

1. Earth Tier 0 + Fire Tier 0 = Lava Tier 1
2. Lava Tier 1 + Water Tier 0 = Obsidian Tier 2



c. Bidirectional

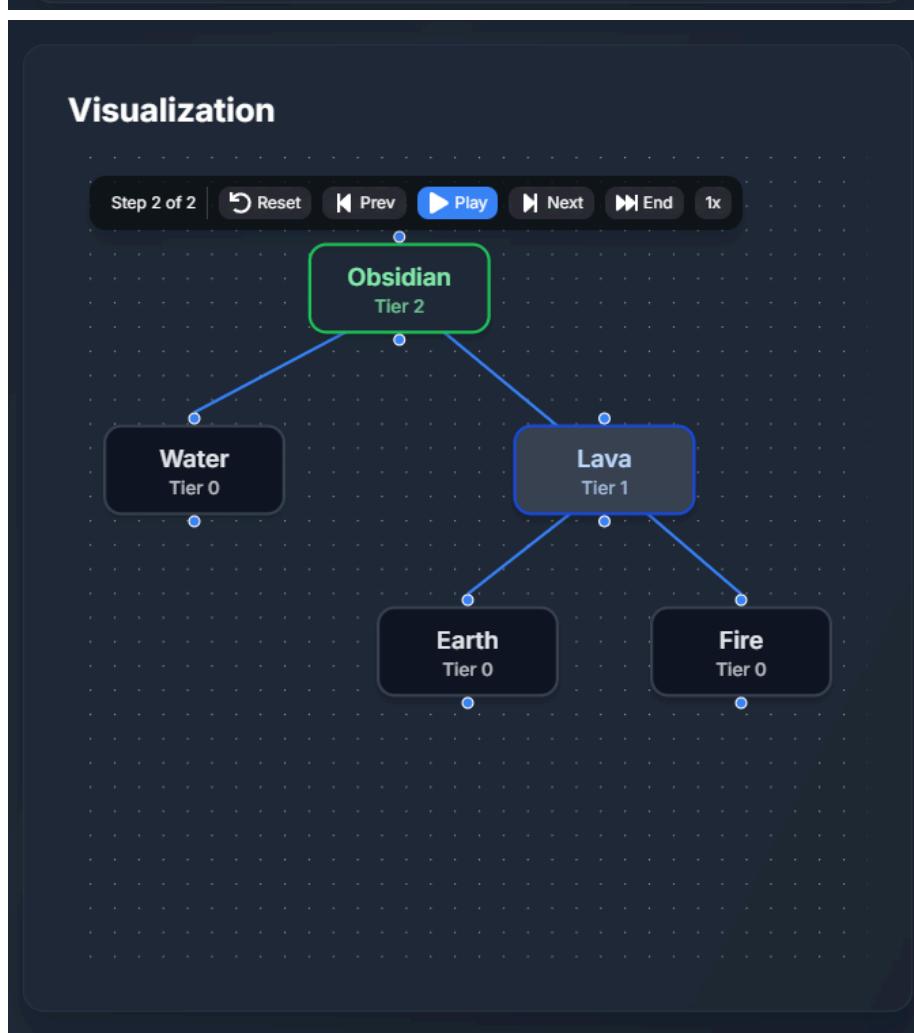
## Search Results

Element found after visiting 39 nodes 0.71 ms

Target: Obsidian Tier 2

**Path:**

1. Earth Tier 0 + Fire Tier 0 = Lava Tier 1
2. Water Tier 0 + Lava Tier 1 = Obsidian Tier 2



## 2. Test Case: Beach

### a. BFS

## Search Results

Element found after visiting 13 nodes

0.17 ms

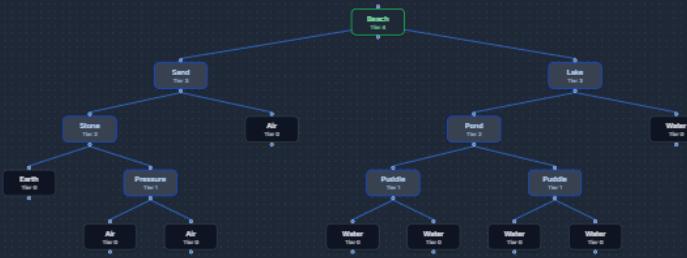
Target: Beach Tier 4

### Path:

1.  
Air Tier 0 + Air Tier 0 = Pressure Tier 1
2.  
Earth Tier 0 + Pressure Tier 1 = Stone Tier 2
3.  
Stone Tier 2 + Air Tier 0 = Sand Tier 3
4.  
Water Tier 0 + Water Tier 0 = Puddle Tier 1
5.  
Water Tier 0 + Water Tier 0 = Puddle Tier 1
6.  
Puddle Tier 1 + Puddle Tier 1 = Pond Tier 2
7.  
Pond Tier 2 + Water Tier 0 = Lake Tier 3
8.  
Sand Tier 3 + Lake Tier 3 = Beach Tier 4

## Visualization

Step 8 of 8 | ⏪ Reset | ⏪ Prev | **Play** | ⏩ Next | ⏩ End | 1x



b. DFS

## Search Results

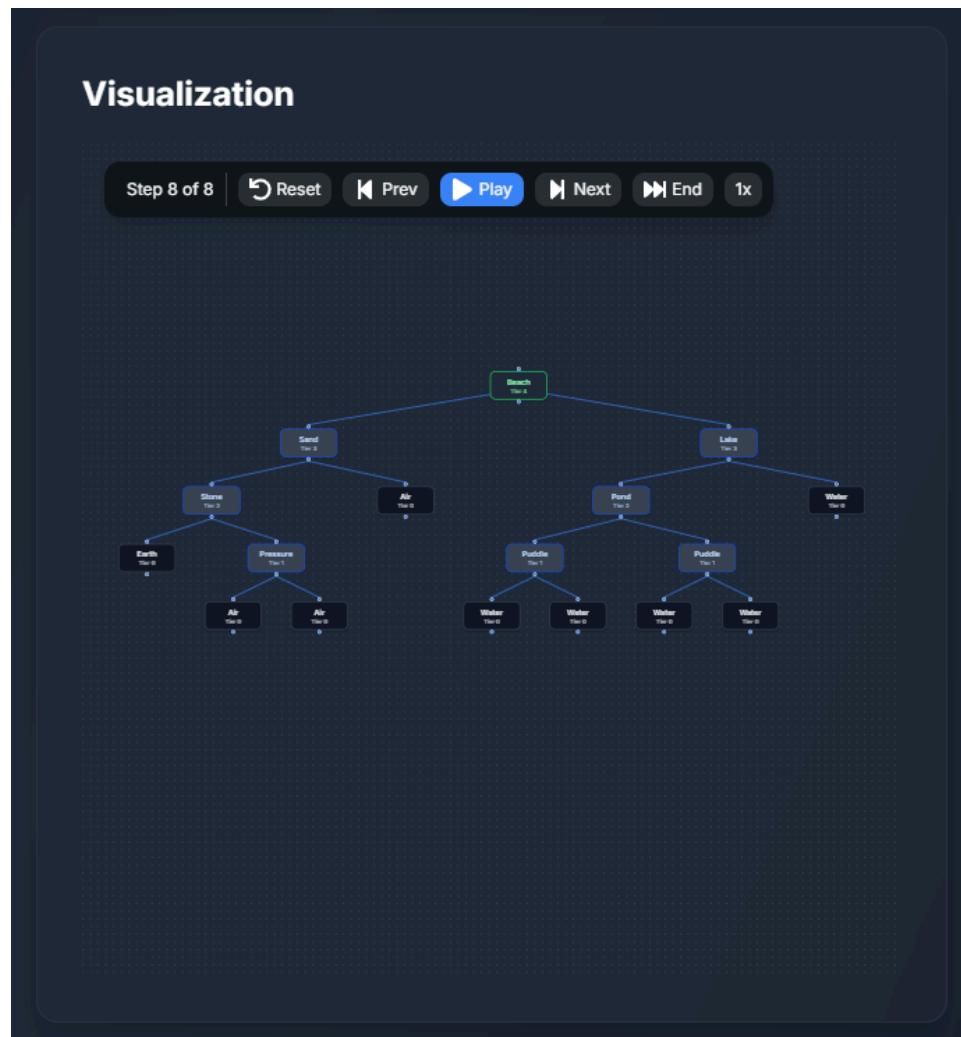
Element found after visiting 17 nodes

0.04 ms

Target: Beach Tier 4

### Path:

1. Air Tier 0 + Air Tier 0 = Pressure Tier 1
2. Earth Tier 0 + Pressure Tier 1 = Stone Tier 2
3. Stone Tier 2 + Air Tier 0 = Sand Tier 3
4. Water Tier 0 + Water Tier 0 = Puddle Tier 1
5. Water Tier 0 + Water Tier 0 = Puddle Tier 1
6. Puddle Tier 1 + Puddle Tier 1 = Pond Tier 2
7. Pond Tier 2 + Water Tier 0 = Lake Tier 3
8. Sand Tier 3 + Lake Tier 3 = Beach Tier 4



c. Bidirectional

## Search Results

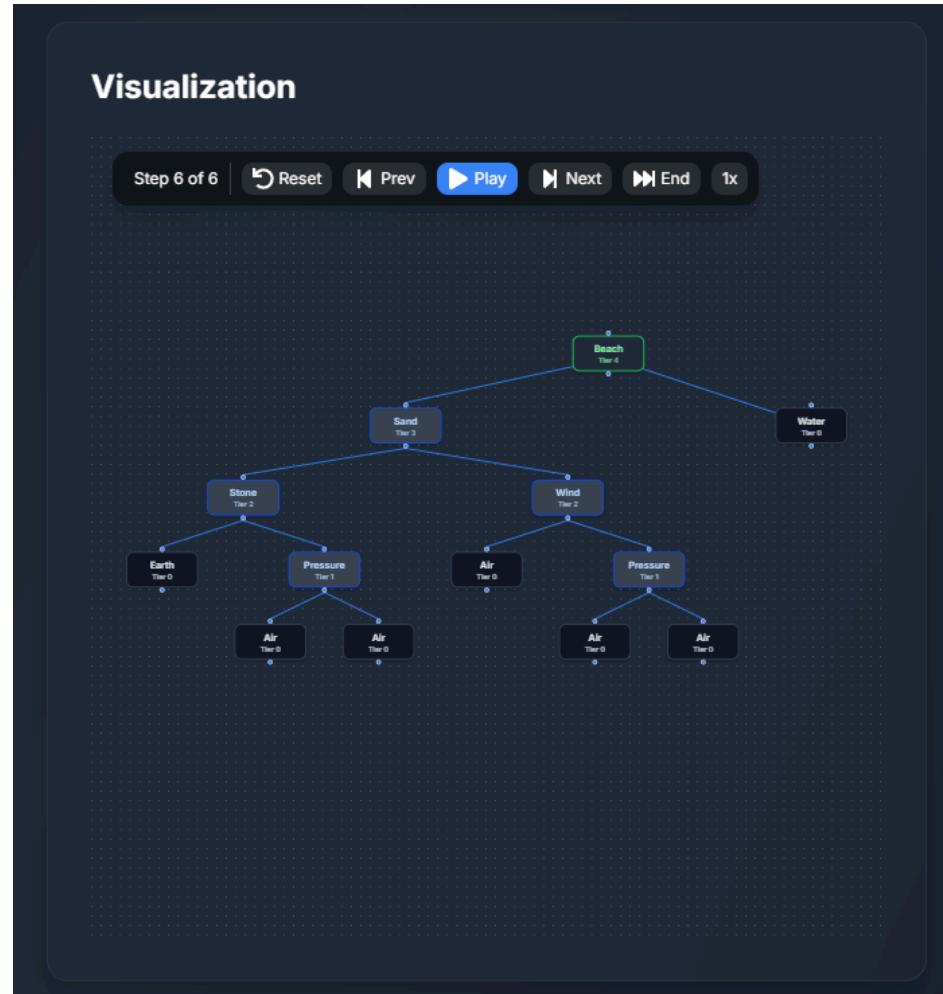
Element found after visiting 123 nodes

2.23 ms

Target: Beach Tier 4

### Path:

1.  
Air Tier 0 + Air Tier 0 = Pressure Tier 1
2.  
Earth Tier 0 + Pressure Tier 1 = Stone Tier 2
3.  
Air Tier 0 + Air Tier 0 = Pressure Tier 1
4.  
Air Tier 0 + Pressure Tier 1 = Wind Tier 2
5.  
Stone Tier 2 + Wind Tier 2 = Sand Tier 3
6.  
Sand Tier 3 + Water Tier 0 = Beach Tier 4



### 3. Test Case: Atmosphere

#### a. BFS

## Search Results

Element found after visiting 6 nodes 2.57 ms

Target: Atmosphere Tier 4

**Path:**

1. Earth Tier 0 + Earth Tier 0 = Land Tier 1
2. Earth Tier 0 + Earth Tier 0 = Land Tier 1
3. Land Tier 1 + Land Tier 1 = Continent Tier 2
4. Earth Tier 0 + Earth Tier 0 = Land Tier 1
5. Earth Tier 0 + Earth Tier 0 = Land Tier 1
6. Land Tier 1 + Land Tier 1 = Continent Tier 2
7. Continent Tier 2 + Continent Tier 2 = Planet Tier 3
8. Air Tier 0 + Planet Tier 3 = Atmosphere Tier 4

**Search Results**

Element found after visiting 6 nodes 2.57 ms

**Target: Atmosphere Tier 4**

**Path:**

1. Earth Tier 0 + Earth Tier 0 = Land Tier 1
2. Earth Tier 0 + Earth Tier 0 = Land Tier 1
3. Land Tier 1 + Land Tier 1 = Continent Tier 2
4. Earth Tier 0 + Earth Tier 0 = Land Tier 1
5. Earth Tier 0 + Earth Tier 0 = Land Tier 1
6. Land Tier 1 + Land Tier 1 = Continent Tier 2
7. Continent Tier 2 + Continent Tier 2 = Planet Tier 3
8. Air Tier 0 + Planet Tier 3 = Atmosphere Tier 4

b. DFS

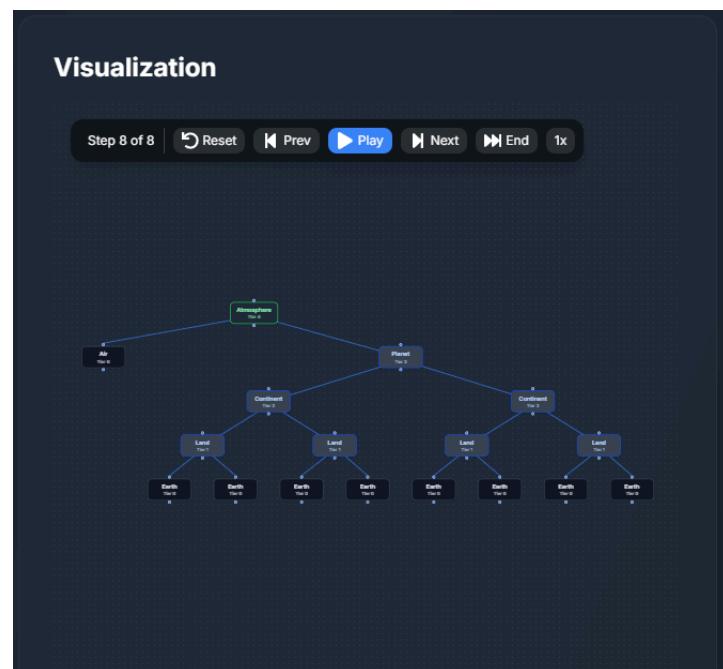
## Search Results

Element found after visiting 17 nodes 0.00 ms

**Target: Atmosphere** Tier 4

**Path:**

1. Earth Tier 0 + Earth Tier 0 = Land Tier 1
2. Earth Tier 0 + Earth Tier 0 = Land Tier 1
3. Land Tier 1 + Land Tier 1 = Continent Tier 2
4. Earth Tier 0 + Earth Tier 0 = Land Tier 1
5. Earth Tier 0 + Earth Tier 0 = Land Tier 1
6. Land Tier 1 + Land Tier 1 = Continent Tier 2
7. Continent Tier 2 + Continent Tier 2 = Planet Tier 3
8. Air Tier 0 + Planet Tier 3 = Atmosphere Tier 4



c. Bidirectional

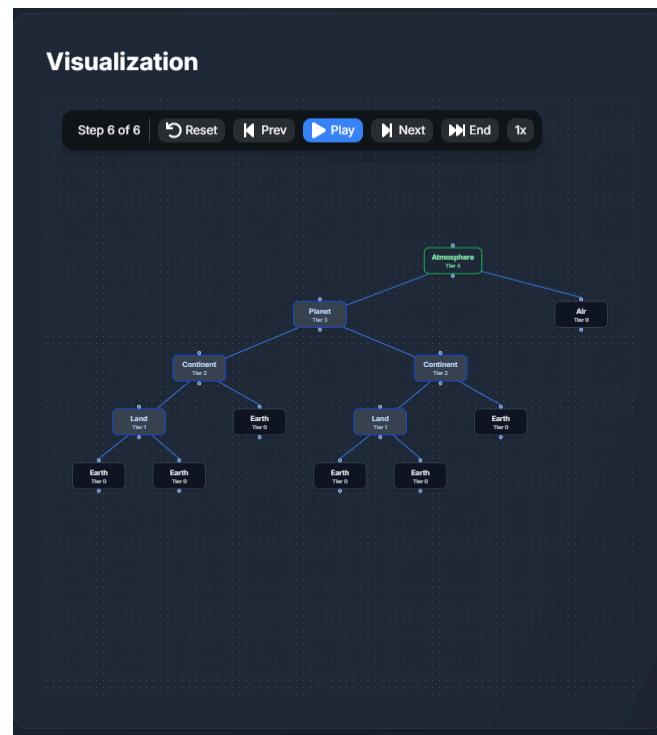
**Search Results**

Element found after visiting 129 nodes 12.61 ms

Target: Atmosphere Tier 4

**Path:**

1. Earth Tier 0 + Earth Tier 0 = Land Tier 1
2. Land Tier 1 + Earth Tier 0 = Continent Tier 2
3. Earth Tier 0 + Earth Tier 0 = Land Tier 1
4. Land Tier 1 + Earth Tier 0 = Continent Tier 2
5. Continent Tier 2 + Continent Tier 2 = Planet Tier 3
6. Planet Tier 3 + Air Tier 0 = Atmosphere Tier 4

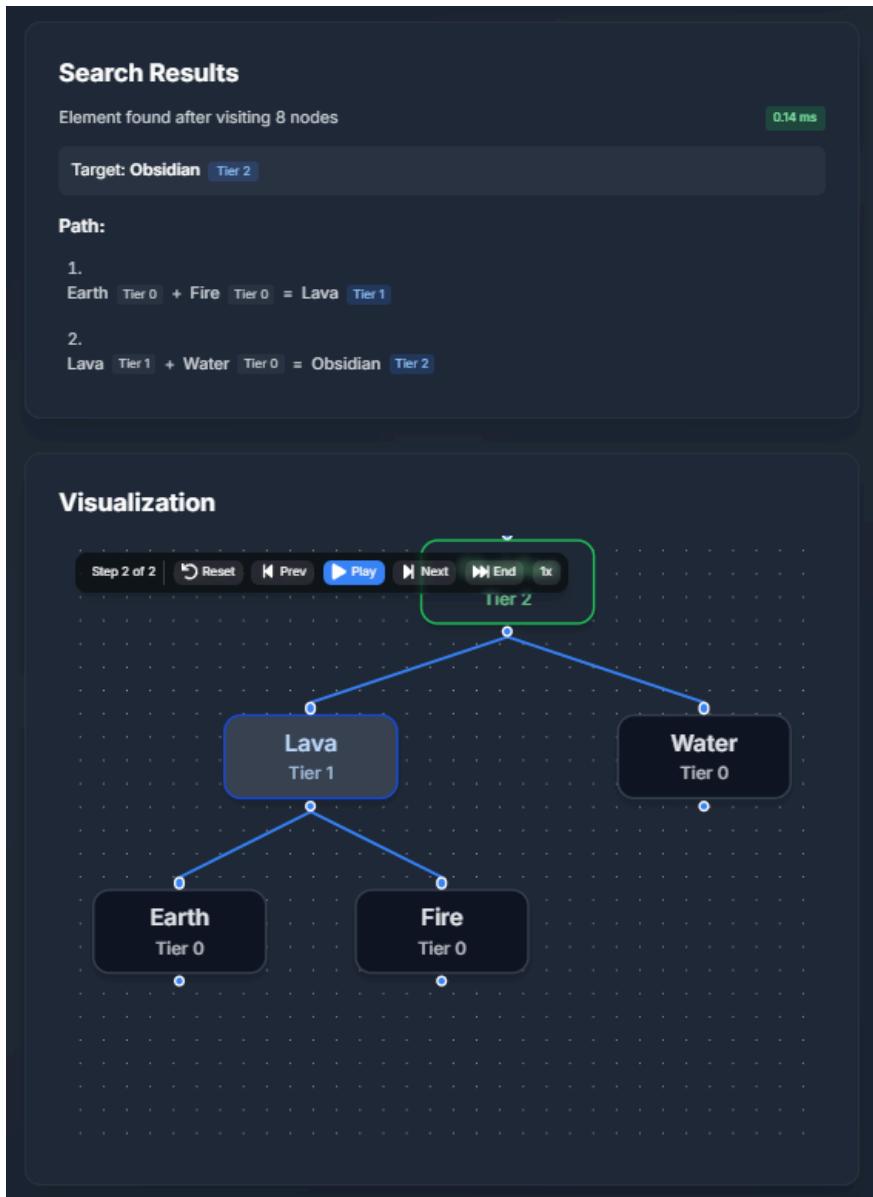


#### 4.3.2 Hasil Pengujian untuk Multi Recipe:

Note: Max Recipe yang digunakan selalu 3.

1. Test Case: Obsidian

a. BFS



b. DFS  
(mirip dengan BFS)

c. Bidirectional  
(mirip dengan BFS)

2. Test Case: Beach

- a. BFS
  - Resep 1

### Search Results

Element found after visiting 61 nodes 0.62 ms

**Target: Beach Tier 4**

**Recipe 1**   **Recipe 2**   **Recipe 3**

**Recipe 1 Path:**

1. Air Tier 0 + Air Tier 0 = Pressure Tier 1
2. Earth Tier 0 + Pressure Tier 1 = Stone Tier 2
3. Stone Tier 2 + Air Tier 0 = Sand Tier 3
4. Sand Tier 3 + Water Tier 0 = Beach Tier 4

### Visualization

Step 4 of 4 | ⏪ Reset | ⏴ Prev | **Play** | ⏵ Next | ⏹ End | ⏹

```

graph TD
    Beach[Beach Tier 4] --> Water[Water Tier 0]
    Beach --> Node1[ ]
    Node1 --> Stone[Stone Tier 2]
    Node1 --> Air1[Air Tier 0]
    Stone --> Earth[Earth Tier 0]
    Stone --> Pressure[Pressure Tier 1]
    Pressure --> Air2[Air Tier 0]
    Pressure --> Air3[Air Tier 0]
  
```

- Resep 2

**Search Results**

Element found after visiting 61 nodes 0.62 ms

Target: Beach Tier 4

Recipe 1 Recipe 2 **Recipe 2** Recipe 3

**Recipe 2 Path:**

1. Earth Tier 0 + Fire Tier 0 = Lava Tier 1
2. Air Tier 0 + Lava Tier 1 = Stone Tier 2
3. Stone Tier 2 + Air Tier 0 = Sand Tier 3
4. Water Tier 0 + Sand Tier 3 = Beach Tier 4

**Visualization**

Step 4 of 4 | Reset | Prev | Play | Next | End | Tx

```

graph TD
    Beach[Beach Tier 4] --> Water[Water Tier 0]
    Beach --> Sand[Sand Tier 3]
    Sand --> Air1[Air Tier 0]
    Sand --> Stone[Stone Tier 2]
    Stone --> Air2[Air Tier 0]
    Stone --> Lava[Lava Tier 1]
    Lava --> Earth[Earth Tier 0]
    Lava --> Fire[Fire Tier 0]
  
```

- Resep 3

## Search Results

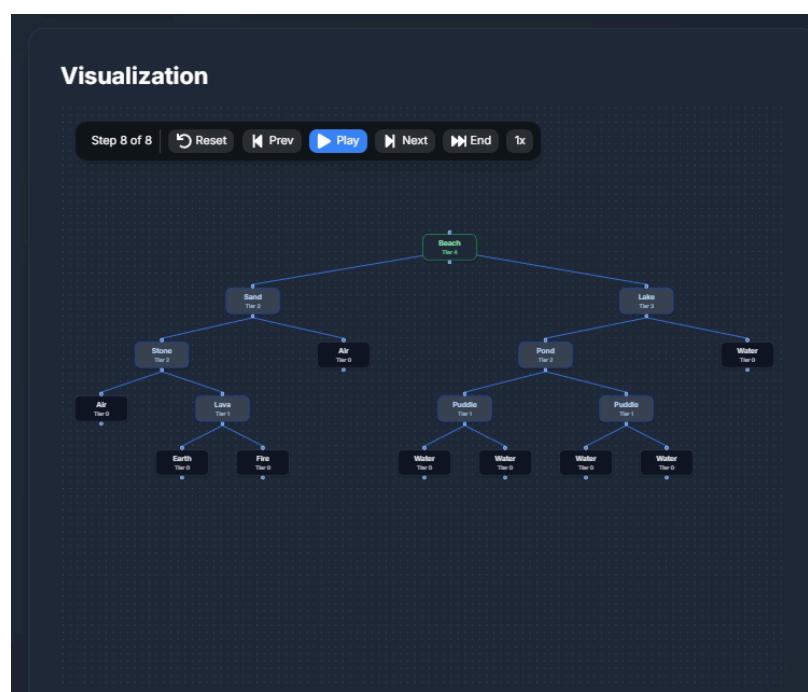
Element found after visiting 61 nodes 0.62 ms

**Target: Beach Tier 4**

**Recipe 1** **Recipe 2** **Recipe 3** (selected)

**Recipe 3 Path:**

1. Earth Tier 0 + Fire Tier 0 = Lava Tier 1
2. Air Tier 0 + Lava Tier 1 = Stone Tier 2
3. Stone Tier 2 + Air Tier 0 = Sand Tier 3
4. Water Tier 0 + Water Tier 0 = Puddle Tier 1
5. Water Tier 0 + Water Tier 0 = Puddle Tier 1
6. Puddle Tier 1 + Puddle Tier 1 = Pond Tier 2
7. Pond Tier 2 + Water Tier 0 = Lake Tier 3
8. Sand Tier 3 + Lake Tier 3 = Beach Tier 4



b. DFS  
(mirip dengan BFS)

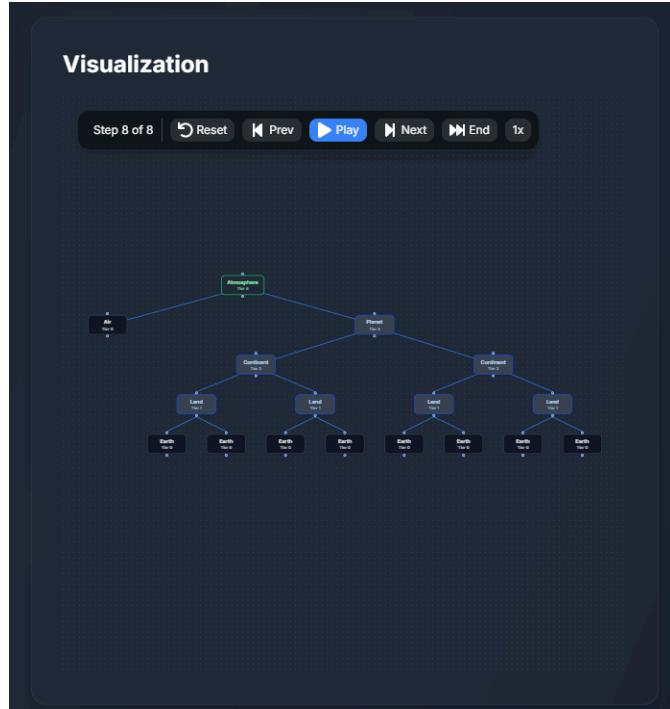
c. Bidirectional  
(mirip dengan BFS)

3. Test Case: Atmosphere

a. BFS

The screenshot displays a search results interface with the following details:

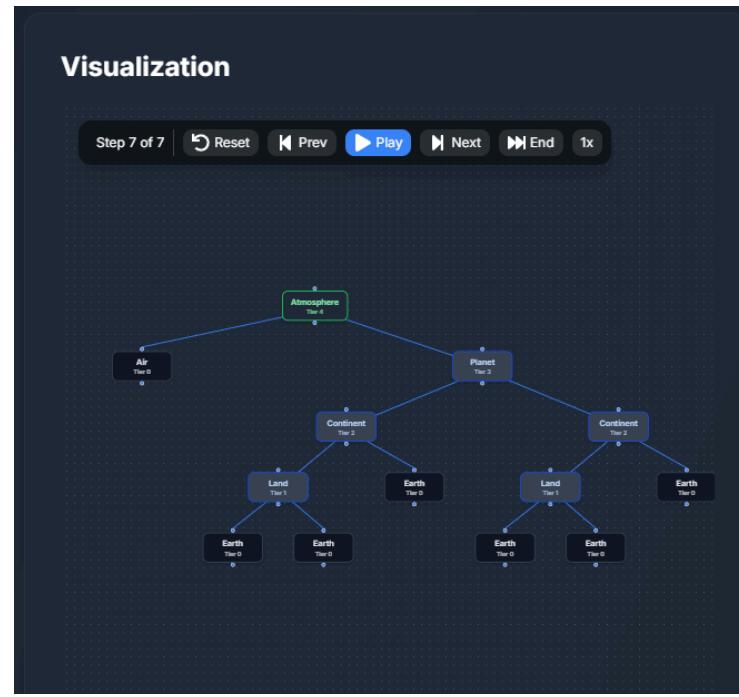
- Search Results**: Element found after visiting 8 nodes (2.10 ms).
- Target**: Atmosphere Tier 4.
- Buttons**: Recipe 1 (highlighted), Recipe 2, Recipe 3.
- Recipe 1 Path:**
  1. Earth Tier 0 + Earth Tier 0 = Land Tier 1
  2. Earth Tier 0 + Earth Tier 0 = Land Tier 1
  3. Land Tier 1 + Land Tier 1 = Continent Tier 2
  4. Earth Tier 0 + Earth Tier 0 = Land Tier 1
  5. Earth Tier 0 + Earth Tier 0 = Land Tier 1
  6. Land Tier 1 + Land Tier 1 = Continent Tier 2
  7. Continent Tier 2 + Continent Tier 2 = Planet Tier 3
  8. Air Tier 0 + Planet Tier 3 = Atmosphere Tier 4



Recipe 1    Recipe 2    Recipe 3

**Recipe 2 Path:**

1. Earth Tier 0 + Earth Tier 0 = Land Tier 1
2. Earth Tier 0 + Earth Tier 0 = Land Tier 1
3. Land Tier 1 + Land Tier 1 = Continent Tier 2
4. Earth Tier 0 + Earth Tier 0 = Land Tier 1
5. Land Tier 1 + Earth Tier 0 = Continent Tier 2
6. Continent Tier 2 + Continent Tier 2 = Planet Tier 3
7. Air Tier 0 + Planet Tier 3 = Atmosphere Tier 4



## Search Results

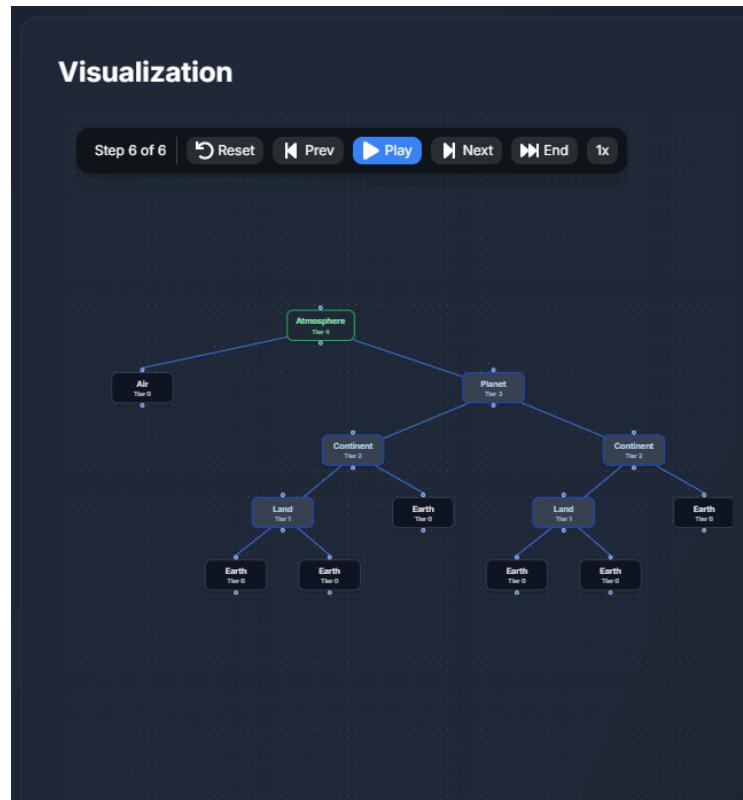
Element found after visiting 8 nodes 2.68 ms

Target: Atmosphere | Tier 4

Recipe 1 Recipe 2 Recipe 3

**Recipe 3 Path:**

1.  $\text{Earth Tier 0} + \text{Earth Tier 0} = \text{Land Tier 1}$
2.  $\text{Land Tier 1} + \text{Earth Tier 0} = \text{Continent Tier 2}$
3.  $\text{Earth Tier 0} + \text{Earth Tier 0} = \text{Land Tier 1}$
4.  $\text{Land Tier 1} + \text{Earth Tier 0} = \text{Continent Tier 2}$
5.  $\text{Continent Tier 2} + \text{Continent Tier 2} = \text{Planet Tier 3}$
6.  $\text{Air Tier 0} + \text{Planet Tier 3} = \text{Atmosphere Tier 4}$



- b. DFS  
(mirip dengan BFS)
- c. Bidirectional  
(mirip dengan BFS)

#### 4.4 Analisis Hasil Pengujian

Bagian ini menganalisis hasil pengujian dari bagian 4.3 untuk mengevaluasi efisiensi dan keterbatasan algoritma resep dalam permainan Little Alchemy 2. Analisis berfokus pada elemen Atmosphere, Obsidian dan Beach untuk pencarian resep tunggal (single-recipe) dan Multi recipes menggunakan algoritma BFS, DFS, dan Bidirectional Search.. Evaluasi mencakup jumlah simpul yang dikunjungi, waktu eksekusi, dan rekomendasi perbaikan.

#### 4.4.1 Analisis Efisiensi Algoritma

Efisiensi diukur berdasarkan jumlah simpul yang dikunjungi, yang mencerminkan beban komputasi algoritma. Analisis dibagi menjadi tiga bagian utama: Obsidian, Beach, dan Atmosphere.

##### 1. Obsidian (Single Recipe)

Tabel 1: Perbandingan Jumlah Simpul yang Dikunjungi dan waktu Eksekusi untuk Elemen Obsidian Single Recipe

Algoritma	Jumlah Simpul Dikunjungi	Waktu Eksekusi (ms)
BFS	5	1.04
DFS	5	0,01
Bidirectional	40	2.65

##### 2. Obsidian (Multi-Recipe).

Tabel 2: Perbandingan Jumlah Simpul yang Dikunjungi dan waktu Eksekusi untuk Elemen Obsidian Multi Recipe

Algoritma	Jumlah Simpul Dikunjungi	Waktu Eksekusi (ms)
BFS	5	1,04
DFS	5	0,50
Bidirectional	2121	124.64

##### 3. Beach (Single Recipe)

Tabel 1: Perbandingan Jumlah Simpul yang Dikunjungi dan waktu Eksekusi untuk Elemen Beach Single Recipe

Algoritma	Jumlah Simpul Dikunjungi	Waktu Eksekusi (ms)

BFS	13	3.72
DFS	17	0,01
Bidirectional	127	11.40

4. Beach (Multi-Recipe).

Tabel 2: Perbandingan Jumlah Simpul yang Dikunjungi dan waktu Eksekusi untuk Elemen Beach Multi Recipe

Algoritma	Jumlah Simpul Dikunjungi	Waktu Eksekusi (ms)
BFS	14	4.27
DFS	28	1,59
Bidirectional	2139	286.58

5. Atmosphere (Single-Recipe).

Tabel 2: Perbandingan Jumlah Simpul yang Dikunjungi dan waktu Eksekusi untuk Elemen Atmosphere Single Recipe

Algoritma	Jumlah Simpul Dikunjungi	Waktu Eksekusi (ms)
BFS	6	2.10
DFS	17	0,01
Bidirectional	129	12.61

6. Atmosphere (Multi-Recipe).

Tabel 2: Perbandingan Jumlah Simpul yang Dikunjungi dan waktu Eksekusi untuk Elemen Atmosphere Multi Recipe

Algoritma	Jumlah Simpul Dikunjungi	Waktu Eksekusi (ms)
BFS	8	2,53
DFS	13	0,61
Bidirectional	1184	171.31

#### 4.4.2 Analisis Performa Algoritma

##### 1. Breadth-First Search (BFS)

BFS menunjukkan performa yang konsisten untuk Single Recipe, dengan jumlah simpul yang dikunjungi relatif rendah (5 untuk Obsidian, 13 untuk Beach, 6 untuk Atmosphere) dan waktu eksekusi yang cepat (1.04–3.72 ms). Pada skenario Multi-Recipe, jumlah simpul yang dikunjungi sedikit meningkat (5–14), tetapi waktu eksekusi tetap efisien (1.04–4.27 ms). Hal ini sesuai dengan sifat BFS yang menjelajahi simpul secara level demi level, sehingga menjamin solusi terpendek dengan overhead minimal untuk elemen dengan tier rendah seperti Obsidian (tier 3), Beach (tier 3), dan Atmosphere (tier 4). BFS sangat efisien untuk pencarian resep terpendek, tetapi tidak dioptimalkan untuk Multi-Recipe karena hanya mencari satu jalur terpendek.

##### 2. Depth-First Search (DFS)

DFS menunjukkan waktu eksekusi yang sangat cepat untuk Single Recipe (0.01 ms untuk semua elemen), dengan jumlah simpul yang dikunjungi sedikit lebih tinggi dibandingkan BFS (5 untuk Obsidian, 17 untuk Beach dan Atmosphere). Pada skenario Multi-Recipe, DFS tetap cepat (0.50–1.59 ms), tetapi jumlah simpul yang dikunjungi meningkat (5–28). DFS efisien dalam hal waktu karena sifatnya yang menelusuri satu jalur hingga selesai sebelum kembali (backtrack), tetapi tidak menjamin solusi terpendek, seperti terlihat pada Beach dan Atmosphere di mana jumlah simpul yang dikunjungi lebih tinggi dibandingkan BFS. Untuk Multi-Recipe, DFS lebih cocok karena dapat menemukan beberapa jalur dengan cepat, meskipun jalur yang ditemukan mungkin tidak optimal.

##### 3. Bidirectional Search

Bidirectional Search menunjukkan performa yang kurang efisien dibandingkan BFS dan DFS, terutama pada skenario Multi-Recipe. Untuk Single Recipe, jumlah simpul yang dikunjungi jauh lebih tinggi (40 untuk Obsidian, 127 untuk Beach, 129 untuk Atmosphere) dengan waktu eksekusi yang lebih lama (2.65–12.61 ms). Pada Multi-Recipe, performa Bidirectional menjadi sangat tidak efisien, dengan jumlah simpul yang dikunjungi mencapai 1184–2139 dan waktu eksekusi 124.64–286.58 ms. Hal ini menunjukkan bahwa implementasi Bidirectional Search saat ini memiliki overhead yang besar, terutama dalam mendeteksi pertemuan simpul dari dua arah pencarian dan mengelola struktur data seperti reverseMap.

#### 4.4.2 Keterbatasan dan Rekomendasi

Berdasarkan analisis, terdapat beberapa keterbatasan:

1. Bidirectional kurang efisien untuk graf kecil, dengan overhead pencarian dua arah.
2. DFS dan Bidirectional memiliki jumlah simpul dikunjungi yang sangat besar, sementara waktu bidirectional bekerja sangat lambat. Ini menunjukkan bahwa untuk DFS dan Bidirectional melakukan eksplorasi yang berlebihan. Berbeda dengan BFS yang lebih sedikit. Tetapi ini tidak selalu benar karena akan bergantung pada besar graf.
3. Walaupun jumlah simpul yang dikunjungi DFS lebih banyak dari BFS (terutama pada multirecipe), DFS lebih cepat menemukan elemen karena BFS kami menggunakan antrian (queue) dan DFS kami menggunakan rekursi (tumpukan implisit melalui stack) yang lebih efisien dalam hal akses memori. Namun untuk teori strategi algoritma itu sendiri, DFS lebih cepat karena DFS langsung menyelam ke cabang pertama yang ditemukan, sedangkan BFS harus memproses semua simpul di lapis yang sama sebelum melangkah ke lapis berikutnya.

Rekomendasi untuk kedepanya:

1. Optimisasi Bidirectional: Kurangi overhead untuk graf kecil dengan deteksi simpul perantara yang lebih efisien, terutama untuk single recipe.
2. Penyempurnaan Multi-Recipe Search: Sesuaikan initialDepth secara dinamis dan tingkatkan serializeTree untuk mengurangi simpul yang dikunjungi, terutama untuk multirecipe.

3. Gunakan DFS untuk graf dengan kedalaman besar dan lebar kecil (sedikit bercabang), dan multithreading dengan Contention Rendah (multirecipe).
4. Gunakan BFS untuk graf dengan lebar besar dan kedalaman kecil (banyak cabang per simpul) dan ingin mencari jalur terpendek.

## BAB 5: Kesimpulan dan Saran

### 5.1 Kesimpulan

Tugas Besar 2 IF2211 Strategi Algoritma ini meminta kami mengimplementasikan algoritma Breadth-First Search (BFS) dan Depth-First Search (DFS) dalam pencarian resep untuk permainan Little Alchemy 2. Kami sudah berhasil membuat website dengan menggunakan bahasa pemrograman GoLang untuk backend yang dapat melakukan pencarian dengan algoritma BFS dan DFS serta JavaScript untuk frontend. Tugas ini memperdalam pemahaman kami tentang strategi algoritma, khususnya dalam pemodelan masalah sebagai graf dan penerapan teknik penelusuran untuk menyelesaikan masalah dunia nyata.

### 5.2 Saran

Berdasarkan pengalaman mengerjakan tugas besar ini, kami memiliki beberapa saran untuk pengembangan lebih lanjut dan tugas di masa depan:

- Peningkatan Fitur: Aplikasi dapat diperluas dengan beberapa fitur seperti penyimpanan resep favorit pengguna, atau visualisasi interaktif yang memungkinkan pengguna mengeksplorasi pohon resep secara dinamis.
- Optimasi Algoritma: Implementasi multirecipe dan bidirectional search saat ini belum sepenuhnya sempurna. Terkadang masih ada beberapa masalah seperti multi recipe yang tidak mendapatkan semua resep, bidirectional yang tidak menemukan resep, dan masalah lainnya.
- Pengelolaan Proyek: Untuk penggeraan tugas selanjutnya, diharapkan untuk lebih meningkatkan komunikasi antar anggota kelompok sehingga dapat meningkatkan efisiensi kerja tim, terutama saat mengintegrasikan frontend dan backend.

### 5.3 Refleksi

Refleksi yang kami dapatkan dari tugas ini adalah tugas ini memperkuat pemahaman kami tentang algoritma penelusuran graf, khususnya BFS dan DFS, serta penerapannya dalam konteks nyata. Kami juga mendapatkan pengalaman berharga dalam pengembangan aplikasi web menggunakan GoLang untuk backend dan JavaScript untuk frontend, serta teknik seperti containerization dengan Docker. Secara non-teknis, kami belajar pentingnya kerja sama tim, pengelolaan waktu, dan komunikasi efektif dalam proyek berkelompok

## LAMPIRAN

- Link Repository : [https://github.com/ivant8k/Tubes2\\_SOS](https://github.com/ivant8k/Tubes2_SOS)
- Link Deploy : <https://alchemix.vercel.app/>
- Link Video Youtube : <https://youtu.be/CD9W-c6la3k?si=SZM8Owx-eOhw8rj7>

### 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**

Rinaldi Munir. Breadth First Search (BFS) dan Depth First Search (DFS) - Bagian

1. Diakses pada 5 Mei 2025 dari

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

Rinaldi Munir. Breadth First Search (BFS) dan Depth First Search (DFS) - Bagian

2. Diakses pada 5 Mei 2025 dari

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-\(2025\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-(2025)-Bagian2.pdf)

Geeksforgeeks. Bidirectional Search. Diakses pada 10 Mei 2025 dari

<https://www.geeksforgeeks.org/bidirectional-search/>