

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

## **Sem II tahun 2024/2025**

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



**Disusun oleh :**

**Salman Hanif - 13523056**

**Guntara Hambali - 13523114**

**Fityatul Haq Rosyidi - 13523116**

**Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung  
2025**

# Daftar Isi

<b>Daftar Isi.....</b>	<b>2</b>
<b>BAB I : Deskripsi Tugas.....</b>	<b>3</b>
<b>BAB II : Landasan Teori.....</b>	<b>5</b>
2.1 Algoritma DFS dan BFS.....	5
2.2 Web Application.....	5
<b>BAB III : Analisis Pemecahan Masalah.....</b>	<b>7</b>
3.1 Langkah-langkah Umum.....	7
3.2 Pemetaan Masalah.....	8
Langkah Pembangunan Tree dengan BFS.....	8
Langkah Pembangunan Tree dengan DFS.....	8
3.3 Arsitektur Aplikasi Web.....	9
3.4 Contoh Ilustrasi kasus.....	10
<b>BAB IV : Implementasi dan Pengujian.....</b>	<b>11</b>
4.1 Spesifikasi teknis program.....	11
4.2 Tata Cara penggunaan program.....	17
4.3 Hasil Pengujian.....	18
4.4 Analisis Hasil Pengujian.....	22
<b>BAB V : Kesimpulan, Saran, dan Refleksi.....</b>	<b>23</b>
5.1 Kesimpulan.....	23
5.2 Saran :v.....	23
5.3 Refleksi.....	24
<b>Lampiran.....</b>	<b>25</b>
<b>Referensi.....</b>	<b>26</b>

# BAB I : Deskripsi Tugas



*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 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 II : Landasan Teori

## 2.1 Algoritma DFS dan BFS

Algoritma Depth-First Search (DFS) adalah metode pencarian dalam struktur data graf, yang bekerja dengan cara menjelajahi satu jalur sedalam mungkin sebelum beralih ke jalur lain. Dalam implementasinya, DFS sering menggunakan struktur data *stack* (tumpukan), baik secara eksplisit maupun implisit melalui rekursi. Algoritma ini dimulai dari simpul awal, lalu menjelajahi simpul-simpul yang terhubung secara mendalam sebelum kembali ke simpul sebelumnya untuk melanjutkan ke jalur lain. Keunggulan DFS terletak pada kemampuannya menemukan solusi tanpa harus memproses semua simpul, terutama dalam graf yang luas tetapi tidak terlalu dalam. Namun, DFS dapat terjebak dalam *loop* jika graf memiliki siklus dan langkah pencegahan seperti pencatatan simpul yang telah dikunjungi tidak diterapkan.

Algoritma Breadth-First Search (BFS), di sisi lain, menjelajahi graf secara bertahap dalam lingkup yang lebih luas sebelum melanjutkan ke tingkat berikutnya. BFS menggunakan struktur data *queue* (antrian) untuk melacak simpul-simpul yang akan dijelajahi berikutnya. Algoritma ini mulai dari simpul awal dan memproses semua simpul yang langsung terhubung dengannya sebelum beralih ke simpul yang lebih jauh. BFS sangat cocok untuk menemukan jalur terpendek dalam graf tak berbobot karena secara sistematis memeriksa semua kemungkinan jalur dengan jarak yang sama terlebih dahulu. Namun, dalam graf yang sangat besar, BFS bisa menjadi tidak efisien karena memerlukan memori yang lebih banyak untuk menyimpan simpul yang akan dieksplorasi.

## 2.2 Web Application

Algoritma Breadth-First Search (BFS) dan Depth-First Search (DFS) yang diterapkan dalam proses pencarian solusi pada permainan *Little Alchemy 2* ini dibungkus dalam bentuk aplikasi berbasis web. Website ini dikembangkan menggunakan framework ReactJS dengan bahasa pemrograman JavaScript XML (JSX). Berdasarkan struktur komponen dalam ReactJS, seluruh tampilan antarmuka web diorganisasi dalam bentuk komponen-komponen terpisah yang saling terintegrasi untuk membentuk sistem yang utuh. Komponen-komponen utama dalam aplikasi ini meliputi Navigation Bar yang berfungsi sebagai judul dan kolom pencarian sekaligus pengontrol input, Main Page sebagai kontainer utama, serta Pop-up Tree yang digunakan untuk menampilkan visualisasi *recipe* dalam bentuk pohon pencarian.

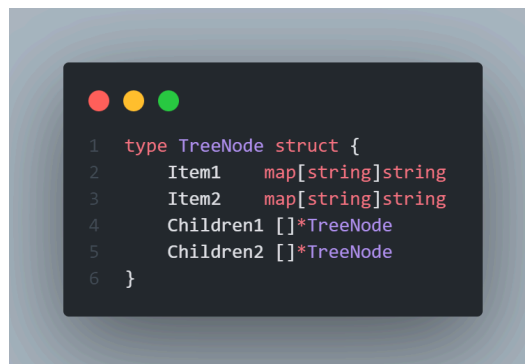
Komponen Main Page terdiri atas elemen Cards dan fitur Pagination. Setiap kartu (card) menampilkan informasi berupa gambar elemen, judul elemen, dan tombol "View" yang digunakan untuk memunculkan Pop-up Tree. Komponen Pop-up Tree sendiri terdiri dari tiga bagian utama, yaitu *Header*, *Main Container*, dan *Footer*. Bagian Header yang bersifat *sticky* mencakup tombol "Close", nama elemen, waktu eksekusi, tombol Zoom In/Out, serta informasi jumlah simpul (nodes) yang telah dikunjungi selama proses pencarian. Bagian *Main Container* digunakan untuk menampilkan struktur tree hasil pencarian secara keseluruhan, sedangkan bagian *Footer* menyediakan fitur pagination untuk menavigasi berbagai kemungkinan *recipe* hasil kombinasi elemen. Struktur modular ini memudahkan pengelolaan kode, pemeliharaan, serta pengembangan fitur lanjutan di masa depan.

# BAB III : Analisis Pemecahan Masalah

## 3.1 Langkah-langkah Umum

Pada website ini, user dapat meminta resep dari suatu elemen dengan memilih metode DFS atau BFS, dan memilih jumlah resep yang ingin dicari. Maka dari itu, FrontEnd akan mengirimkan tuple berupa nama elemen, metode, dan jumlah resep. Di BackEnd, Dilakukan scraping data resep-resep yang tersedia dari website resmi Little Alchemist 2. Scraping ini akan menghasilkan map dengan key nya adalah suatu string nama elemen dan valuenya adalah list of tuple string (daftar pasangan elemen pembentuk key). Nah dari sini, akan dibentuk pohon resep-resep yang dapat membentuk elemen yang tadi diminta. Jika metode yang diminta adalah BFS, pohon resep akan dibentuk dengan menggunakan Queue, dibentuk berurutan dari level teratas menuju level terbawah. Sedangkan jika metode yang diminta adalah DFS, pohon resep dibentuk dengan cara rekursif (masuk ke dalam dulu).

Setelah pohon resep nya dibentuk, akan dilakukan dekomposisi untuk menghasilkan pohon-pohon resep unik. Pohon resep unik adalah pohon resep yang mengandung tepat satu resep. Agar lebih paham, berikut adalah struktur pohon yang kami desain



```
1 type TreeNode struct {
2     Item1    map[string]string
3     Item2    map[string]string
4     Children1 []*TreeNode
5     Children2 []*TreeNode
6 }
```

Gambar 3 Struktur data TreeNode

Treenode diatas menggambarkan pasangan elemen. Item 1 dan 2 berisi nama elemen dan nama gambar. Children 1 dan 2 berisi list dari pasangan-pasangan lain yang membentuk item 1 dan 2 berurutan. Children 1 dan 2 dapat berisi banyak pasangan. Nah, Pohon resep unik adalah pohon resep yang Item 1 dan Item 2 nya masing-masing memiliki tepat 1 Children saja, ataupun tidak memiliki Children (base element).

Dekomposisi akan menghasilkan list of pohon resep unik yang nantinya akan dikirim ke frontend bersamaan dengan waktu eksekusi dan jumlah node yang dikunjungi. Secara bertahap, langkah-langkah algoritmanya adalah sebagai berikut :

1. BackEnd menerima elemen yang diminta, berserta metode dan jumlah node maksimal yang ingin dicari
2. Pohon resep dibangun berdasarkan metode yang diminta (DFS/BFS)
3. Pohon resep yang dihasilkan akan didekomposisi menjadi pohon-pohon resep unik dan dimasukkan ke suatu list
4. List dikirim ke FrontEnd bersama dengan waktu eksekusi dan jumlah node yang dikunjungi

### **3.2 Pemetaan Masalah**

#### **Langkah Pembangunan Tree dengan BFS**

1. Pertama-tama, dibuat queue string dengan 1 anggota yaitu elemen yang dicari. Queue digunakan untuk menampung antrian elemen yang akan disimpan
2. Dilakukan iterasi sampai queue menjadi kosong
3. Setiap iterasi dilakukan dequeue dari pasangan elemen paling depan
4. Untuk setiap pasangan (kanan dan kiri) akan dicari pasangan-pasangan pembentuk elemen tersebut dari hasil scraping, seterusnya akan disebut sebagai children element
5. Jika children element adalah base element atau time, lanjutkan ke iterasi berikutnya
6. Jika tier children element lebih tinggi atau sama dengan element saat ini, lanjutkan
7. Tree baru dibentuk, masukkan ke children tree iterasi saat ini. kemudian Tree baru tersebut masukkan ke dalam queue
8. Iterasi selesai jika queue habis

#### **Langkah Pembangunan Tree dengan DFS**

1. Pembangunan tree dengan DFS dilakukan dengan cara rekursif
2. Pertama-tama, fungsi menerima parameter tree yang mengandung elemen yang dicari (karena tree membutuhkan 2 elemen, dipasangkan dengan nullTree)
3. Untuk setiap item<sub>1</sub> dan item<sub>2</sub>, akan dicari elemen-elemen pembentuknya (children element)



4. Jika children element adalah base element atau time, lanjutkan ke iterasi berikutnya
5. Jika tier children element lebih tinggi atau sama dengan element saat ini, lanjutkan
6. Tree baru dibentuk, sambungkan sebagai children
7. Kemudian fungsi rekursi dipanggil dengan parameter tree yang baru dibentuk

### **3.3 Arsitektur Aplikasi Web**

Aplikasi ini menggunakan arsitektur client-server, sistem dibagi menjadi dua komponen utama : frontend (client) dan backend (server). Komunikasi antara keduanya dilakukan melalui protokol HTTP menggunakan REST API.

#### **FRONT END**

- Dibangun menggunakan ReactJS dengan bundler modern Vite, yang menyediakan kecepatan build dan pengembangan yang optimal.
- Pengguna memasukkan input elemen resep, memilih metode pencarian (BFS atau DFS), serta mengatur batas pencarian ( jumlah node maksimal).
- Setelah input diberikan, frontend akan mengirim permintaan (request) ke server menggunakan fetch API.

#### **BACK END**

- Dibangun menggunakan bahasa pemrograman Go (Golang) yang mendukung multithreading.
- Menyediakan endpoint API yang menerima permintaan pencarian tree dari klien.
- Backend akan membangun tree solusi resep berdasarkan input dan algoritma yang dipilih, lalu mengembalikan: list tree resep, waktu komputasi, dan jumlah node yang dikunjungi selama pencarian

#### **Komunikasi Client-Server**

- Klien berkomunikasi dengan server HTTP request-response cycle.
- Klien mengirim permintaan dengan method post ke endpoint yang sesuai.
- Server memproses permintaan, lalu mengirim respon dalam format JSON yang mudah dibaca dan diproses kembali oleh klien.
- Arsitektur ini mendukung pemisahan tanggung jawab yang jelas antara tampilan (frontend) dan logika pemrosesan data (backend).

### 3.4 Contoh Ilustrasi kasus

Misalkan kasus ketika pengguna meminta resep dari elemen Tsunami. Tsunami dapat dibentuk oleh Sea + Earthquake. Sea dapat dibentuk oleh Lake + Lake. Earthquake dapat dibentuk oleh Earth+Energy. Lake dapat dibentuk dari Pond+Water dan Pond+Pond. Pond dapat dibentuk dari Puddle+Water, Energy dapat dibentuk oleh Fire+Fire

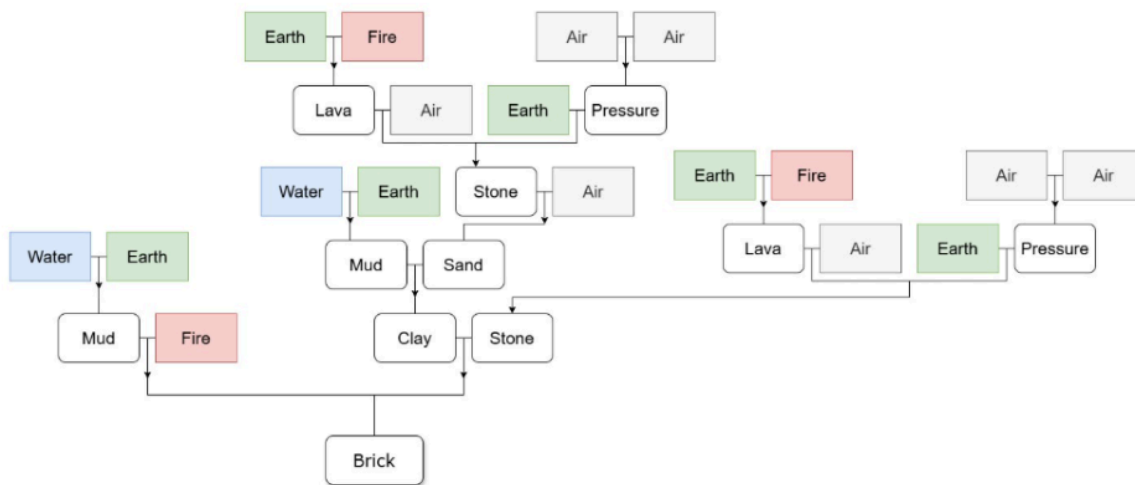
Untuk Algoritma BFS, Tree dibentuk dengan urutan Tsunami => Sea+Earthquake => Lake+Lake => Earth+Energy => Pond+Pond => Pond+Water => Fire+Fire => dst.

Sedangkan untuk Algoritma DFS, Tree dibentuk dengan urutan Tsunami => Sea+Earthquake => Lake+Lake => Pond+Water => Puddle + Puddle => Water+Water => Water+Water => Pond+Pond => Puddle+Puddle=>Water+Water => Water+Water =>Puddle+Water => Water + Water => Earth+Energy => Fire+Fire

# BAB IV : Implementasi dan Pengujian

## 4.1 Spesifikasi teknis program

Untuk memodelkan permasalahan, kami menggunakan struktur data tree dengan sebuah node tree merepresentasikan gabungan 2 element. Pada node juga terdapat daftar children dari masing-masing elemen. Children ini merepresentasikan pasangan pasangan elemen yang dapat membentuk elemen tersebut. Berikut adalah salah satu contoh tree



Gambar 4 Visualisasi tree recipe

Berdasarkan contoh tree diatas, dapat dibentuk tree yang memiliki elemen Brick dan Null, Brick memiliki children berupa 2 buah tree yaitu Clay-Stone dan Mud-Fire. Dan Null tidak memiliki children. lalu clay memiliki 1 children dan stone memiliki 2 children. Mud memiliki 1 children dan Fire tidak memiliki children.

Implementasi struktur data tree tersebut kami bentuk dengan kode seperti berikut

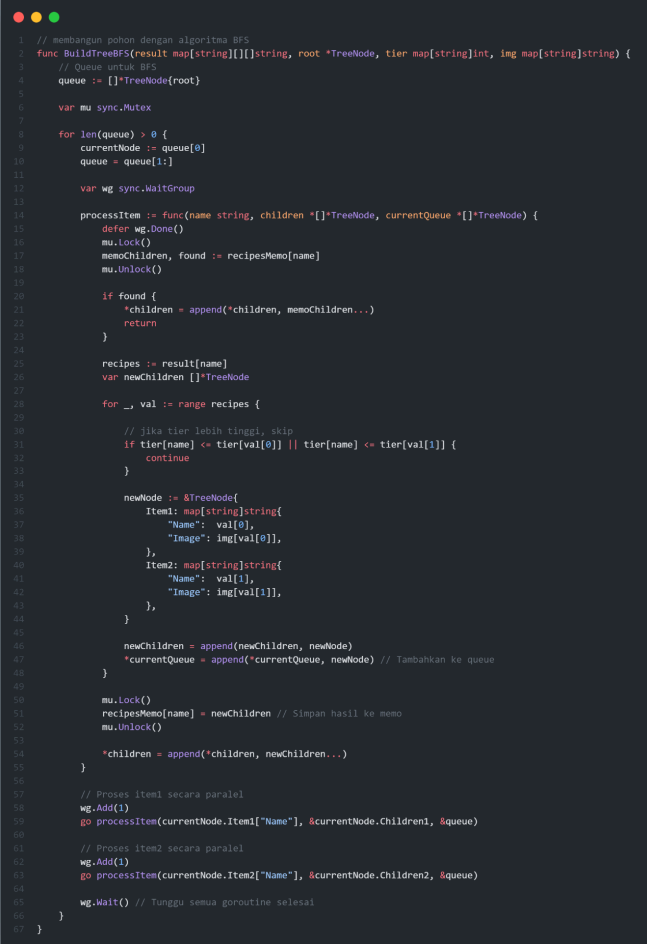
```
1 type TreeNode struct {
2     Item1  map[string]string
3     Item2  map[string]string
4     Children1 []*TreeNode
5     Children2 []*TreeNode
6 }
```

Gambar 5 Struktur data TreeNode

Item 1 dan 2 berisi string nama elemen dari nama image dari elemen tersebut. Children 1 dan 2 adalah list dari tree yang dapat membentuk item 1 dan 2 secara berurutan.

Pada program kami di bagian BackEnd, terdapat 3 package : scrape, utils, dan main. Fungsi dan prosedur di package scrape digunakan untuk melakukan scraping nama elemen, tier, dan path gambar elemen. fungsi dan prosedur di package utils digunakan untuk melakukan pembangunan tree, dekomposisi tree, read dan save tree menjadi json, fungsi filter, fungsi converter, dan mekanisme memoisasi. Fungsi dan prosedur di package main adalah fungsi main itu sendiri.

Ada 4 Fungsi dan Prosedur utama pada program ini :

No	Snapshot kode	Penjelasan
1.	 <pre> 1 // membangun pohon dengan algoritma BFS 2 func BuildTreeBFS(result map[string][][]string, root *TreeNode, tier map[string]int, img map[string]string) { 3     // queue untuk BFS 4     queue := []*TreeNode{root} 5 6     var mu sync.Mutex 7 8     for len(queue) &gt; 0 { 9         currentNode := queue[0] 10        queue = queue[1:] 11 12        var wg sync.WaitGroup 13 14        processItem := func(name string, children []*TreeNode, currentQueue []*TreeNode) { 15            defer wg.Done() 16            mu.Lock() 17            memoChildren, found := recipesMemo[name] 18            mu.Unlock() 19 20            if found { 21                *children = append(*children, memoChildren...) 22                return 23            } 24 25            recipes := result[name] 26            var newChildren []*TreeNode 27 28            for _, val := range recipes { 29                // jika tier lebih tinggi, skip 30                if tier[name] &lt;= tier[val[0]]    tier[name] &lt;= tier[val[1]] { 31                    continue 32                } 33 34                newNode := &amp;TreeNode{ 35                    Item1: map[string]string{ 36                        "Name": val[0], 37                        "Image": img[val[0]], 38                    }, 39                    Item2: map[string]string{ 40                        "Name": val[1], 41                        "Image": img[val[1]], 42                    }, 43                } 44 45                newChildren = append(newChildren, newNode) 46                *currentQueue = append(*currentQueue, newNode) // Tambahkan ke queue 47            } 48 49            mu.Lock() 50            recipesMemo[name] = newChildren // Simpan hasil ke memo 51            mu.Unlock() 52 53            *children = append(*children, newChildren...) 54        } 55 56        // Proses item1 secara paralel 57        wg.Add(1) 58        go processItem(currentNode.Item1["Name"], &amp;currentNode.Children1, &amp;queue) 59 60        // Proses item2 secara paralel 61        wg.Add(1) 62        go processItem(currentNode.Item2["Name"], &amp;currentNode.Children2, &amp;queue) 63 64        wg.Wait() // Tunggu semua goroutine selesai 65    } 66 } </pre>	<p>Prosedur ini membangun tree menggunakan algoritma BFS. fungsi ini juga dioptimasi dengan menggunakan teknik memoisasi dan multithreading. Prosedur ini berada di file BFS.go</p>

2.

```
1 // membangun pohon dengan algoritma DFS
2 func BuildTreeDFS(result map[string][]string, root *TreeNode, tier map[string]int, img map[string]string) {
3     var memoMutex sync.Mutex
4
5     var processItem func(name string, children []*TreeNode, tier map[string]int, wg *sync.WaitGroup)
6     processItem = func(name string, children []*TreeNode, tier map[string]int, wg *sync.WaitGroup) {
7         defer wg.Done() /
8
9         memoMutex.Lock()
10        memoChildren, found := recipesMemo[name]
11        memoMutex.Unlock()
12
13        if found {
14            *children = append(*children, memoChildren...) // Ambil dari memo jika sudah diproses
15            return
16        }
17
18        recipes := result[name]
19        var newChildren []*TreeNode
20
21        for _, val := range recipes {
22
23            // jika tier lebih tinggi, skip
24            if tier[name] <= tier[val[0]] || tier[name] <= tier[val[1]] {
25                continue
26            }
27
28            newNode := &TreeNode{
29                Item1: map[string]string{
30                    "Name": val[0],
31                    "Image": img[val[0]],
32                },
33                Item2: map[string]string{
34                    "Name": val[1],
35                    "Image": img[val[1]],
36                },
37            }
38            newChildren = append(newChildren, newNode)
39
40            wg.Add(1)
41            go processItem(val[0], &newNode.Children1, tier, wg)
42            wg.Add(1)
43            go processItem(val[1], &newNode.Children2, tier, wg)
44        }
45
46        memoMutex.Lock()
47        recipesMemo[name] = newChildren
48        memoMutex.Unlock()
49
50        *children = append(*children, newChildren...)
51    }
52
53    // Proses Item1 dan Item2 secara paralel
54    var wg sync.WaitGroup
55
56    wg.Add(1)
57    go processItem(root.Item1["Name"], &root.Children1, tier, &wg)
58
59    wg.Add(1)
60    go processItem(root.Item2["Name"], &root.Children2, tier, &wg)
61
62    wg.Wait() // Tunggu semua goroutine selesai
63 }
64
65 }
```

Prosedur ini membangun tree menggunakan algoritma DFS. fungsi ini juga dioptimasi dengan menggunakan teknik memoisasi dan multithreading. Prosedur ini berada di file DFS.go

3.

```

1 // menghasilkan seluruh resep
2 func GenerateRecipesTree(node *TreeNode, countRecipe int) ([]*TreeNode, int) {
3     if node == nil {
4         return nil, 0
5     }
6
7     // cek apakah node adalah leaf
8     if len(node.Children1) == 0 && len(node.Children2) == 0 {
9         return []*TreeNode{node}, 1
10    }
11    if (len(node.Children1) == 0) {
12        node.Children1 = append(node.Children1, NullChild) // jika tidak memiliki children1, padding dengan nullChild
13    }
14    if (len(node.Children2) == 0) {
15        node.Children2 = append(node.Children2, NullChild) // jika tidak memiliki children2, padding dengan nullChild
16    }
17
18    var combinations, childCombinations1, childCombinations2 []*TreeNode
19    var nodeCount1, nodeCount2 int
20
21    count := 1
22    nodeCount := 0
23
24    for _, child1 := range node.Children1 {
25
26        key1 := GenerateRecipesMemoKey(child1)
27        if r1, found := recipesMemo[key1]; found {
28            childCombinations1 = r1
29        } else {
30
31            childCombinations1, nodeCount1 = GenerateRecipesTree(child1, countRecipe)
32            nodeCount += nodeCount1
33            recipesMemo[key1] = childCombinations1
34        }
35
36        for _, child2 := range node.Children2 {
37
38            keyComb := GenerateCombinationsMemoKey(child1, child2)
39
40            if savedComb, found := combinationsMemo[keyComb]; found {
41                combinations = append(combinations, savedComb...)
42            } else {
43                break
44            }
45
46            key2 := GenerateRecipesMemoKey(child2)
47            if r2, found := recipesMemo[key2]; found {
48                childCombinations2 = r2
49            } else {
50
51                childCombinations2, nodeCount2 = GenerateRecipesTree(child2, countRecipe)
52                nodeCount += nodeCount2
53                recipesMemo[key2] = childCombinations2
54            }
55
56            var tempCombinations = []*TreeNode{}
57            for _, subTree1 := range childCombinations1 {
58                for _, subTree2 := range childCombinations2 {
59
60                    combinedTree := &TreeNode{
61                        Item1: map[string]string{"Name": node.Item1["Name"], "Image": node.Item1["Image"]},
62                        Item2: map[string]string{"Name": node.Item2["Name"], "Image": node.Item2["Image"]},
63                        Children1: []*TreeNode{subTree1},
64                        Children2: []*TreeNode{subTree2},
65                    }
66
67                    tempCombinations = append(tempCombinations, combinedTree)
68                }
69            }
70
71            combinations = append(combinations, combinedTree)
72            count++
73        }
74    }
75
76    return combinations, count
77 }

```

GenerateRecipesTree() adalah fungsi yang melakukan dekomposisi tree dari tree yang mengandung banyak kombinasi resep menjadi tree-tree dengan resep unik. Jumlah tree-tree resep unik yang ingin dibuat dapat ditentukan berdasarkan parameter countRecipe. Fungsi ini berada di file Generator.go

4.

```
1 func GetRecipes(name string, method int, maxRecipe int) ([]*TreeNode, int, error) {
2     // 0 : BFS
3     // 1 : DFS
4
5     ClearMemo()
6
7     // membaca recipe global
8     globalRecipes, err := ReadElementsRecipes("data/allElementsRecipes.json")
9     if err != nil {
10         log.Fatalf("Failed to read recipes: %v", err)
11     }
12
13     // membaca tier elemen
14     tier, err := ReadElementsTier("data/allElementsTiers.json")
15     if err != nil {
16         log.Fatalf("Failed to read tier: %v", err)
17     }
18
19     // membaca image elemen
20     img, err := ReadElementsImage("data/images.json")
21     if err != nil {
22         log.Fatalf("Failed to read tier: %v", err)
23     }
24
25     // buat root tree
26     root := &TreeNode{
27         Item1: map[string]string{"Name": name},
28         Item2: map[string]string{"Name": "Null"},
29         Children2: []*TreeNode{NullChild},
30     }
31
32     // pilih method untuk membangun pohon resep
33     if (method == 0) {
34         BuildTreeBFS(globalRecipes, root, tier, img)
35     } else if (method == 1) {
36         BuildTreeDFS(globalRecipes, root, tier, img)
37     } else {
38         log.Fatalf("Wrong method!")
39     }
40
41     recipes, nodeCount := GenerateRecipesTree(root, maxRecipe)
42     for _, recipe := range recipes {
43         FilterNullChildren(recipe) // filter anak yang null
44     }
45     recipes = FilterAllParents(recipes)
46
47     return recipes, nodeCount, err
48 }
49
50 }
```

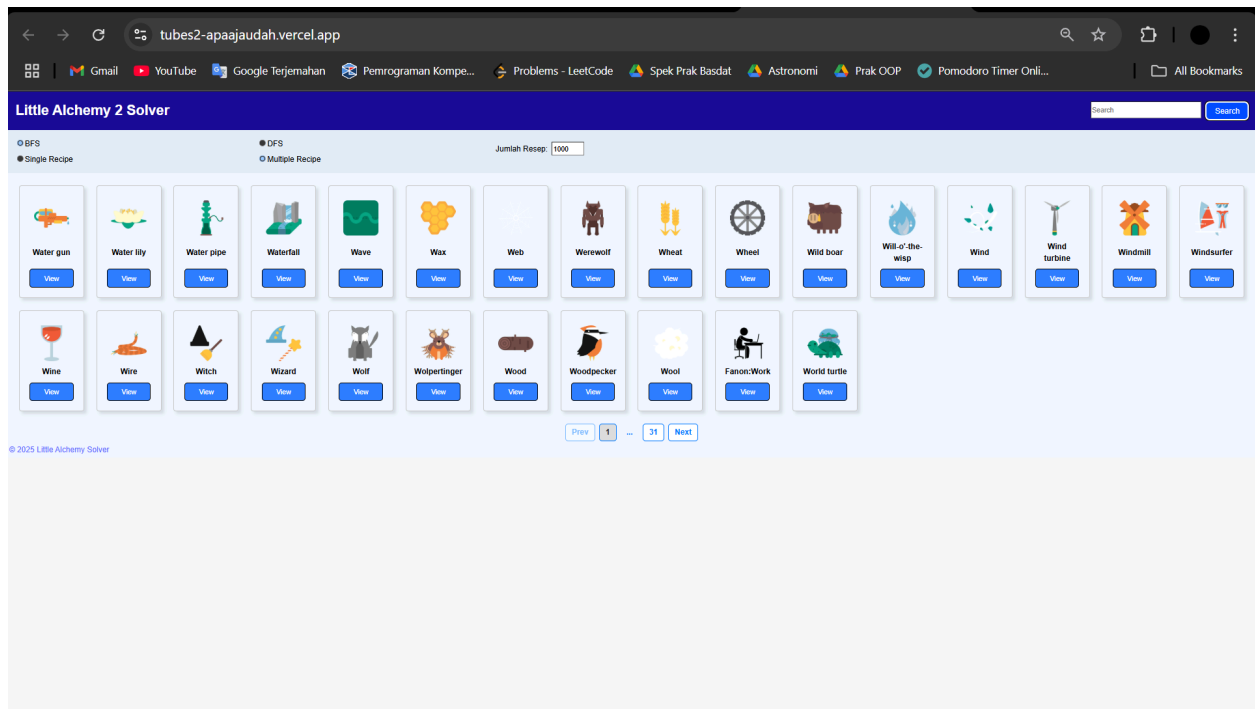
GetRecipes() adalah fungsi yang memanggil ketiga fungsi utama diatas. Fungsi ini membaca semua data-data json yang diperlukan seperti data nama, tier, dan resep-resep dari semua elemen yang ada. Tree root juga dibentuk disini yang kemudian akan di pass ke fungsi pembangunan tree. Setelah tree dibangun, tree didekomposisi dan list hasilnya akan di return oleh fungsi ini bersamaan dengan jumlah node yang dikunjungi. Nantinya fungsi inilah yang akan dipanggil di Main. Fungsi ini berada di file Generator.go

Selain fungsi dan prosedur utama, ada juga fungsi dan prosedur helper yang digunakan di program sebagai pembantu dan pelengkap. Seperti fungsi filtering yang memfilter nullTree (filter.go), fungsi IO yang membaca dan menyimpan file json (io.go), dan fungsi-fungsi utilitas tree (tree.go)



## 4.2 Tata Cara penggunaan program

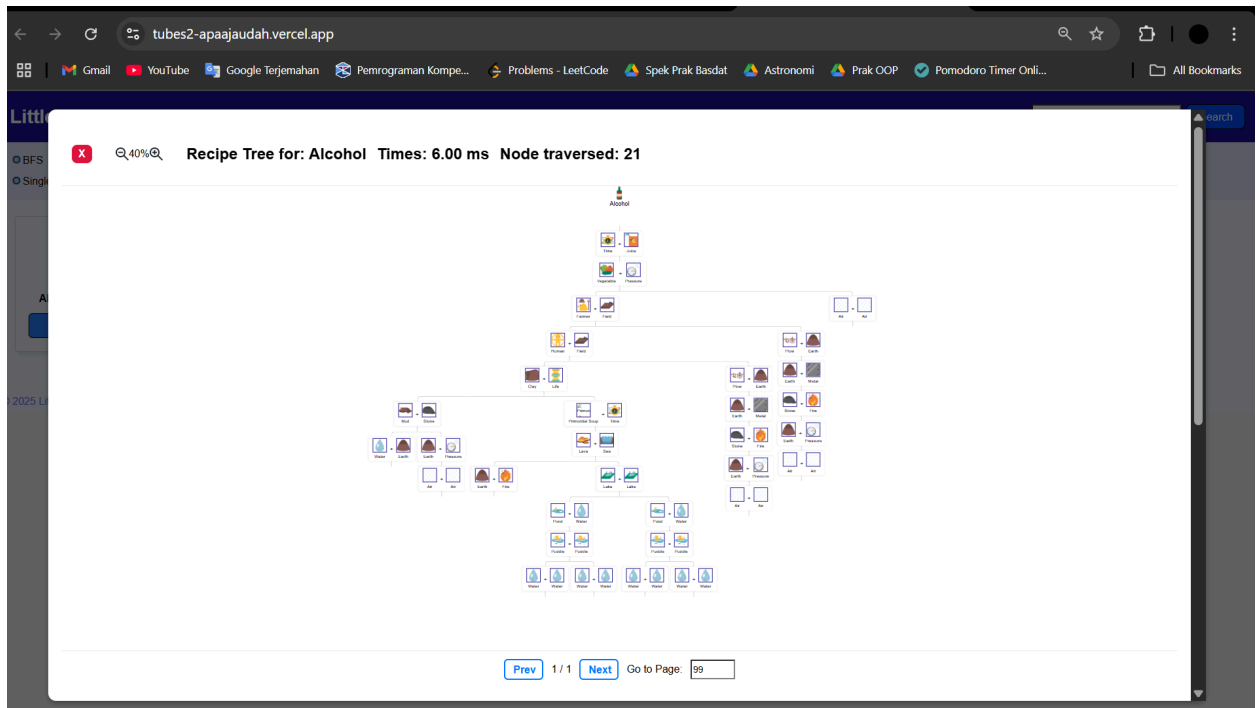
Berikut merupakan tampilan awal program.



Gambar 6 Landing page

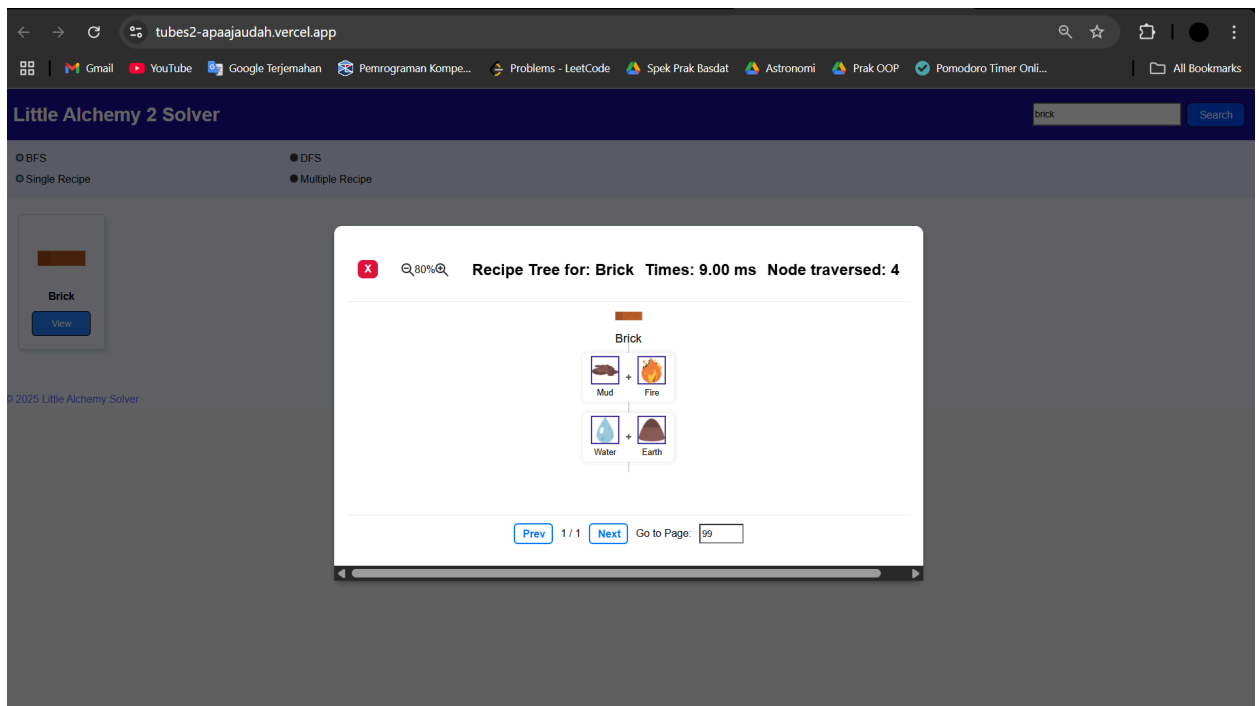
Di kiri atas, ada button untuk memilih mode pencarian (BFS atau DFS). Lalu, pengguna juga dapat memilih ingin mencari 1 recipe atau lebih. Jika multiple recipe, pengguna harus memasukkan recipe maksimal yang akan ditemukan. Di sebelah kanan atas, terdapat search box untuk mencari elemen (dari element yang sangat banyak). Lalu, setelah menemukan elemen yang ingin dicari recipe nya, pengguna tinggal menekan tombol View pada elemen tersebut.





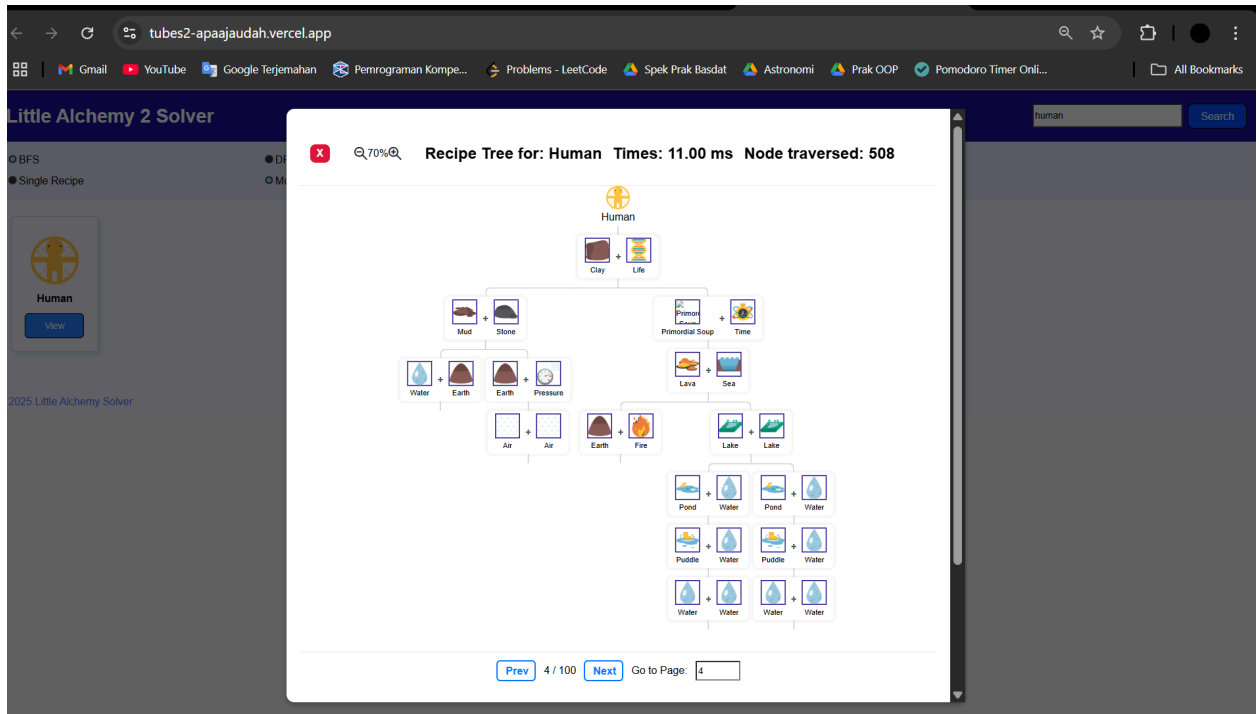
Gambar 7 Pengujian elemen Alchohol mode BFS single recipe

- Pencarian elemen Brick dengan DFS single recipe:



Gambar 8 Pengujian elemen Brick mode DFS single recipe

- Pencarian elemen Human dengan BFS multiple recipe = 100



Gambar 9 Pengujian elemen Human mode BFS single recipe multiple recipe = 100

- Pencarian elemen Werewolf dengan BFS multiple recipe = 1000



#### **4.4 Analisis Hasil Pengujian**

Dari 5 pengujian yang dilakukan di atas, tampak bahwa hasil sesuai yang diharapkan. Waktu yang diperlukan berada di kisaran belasan ms, bahkan untuk multiple recipe. Hasil yang cukup cepat ini bisa diperoleh karena dimanfaatkan konsep memoisasi. Konsep memoisasi bertujuan agar tidak perlu dilakukan komputasi ulang untuk elemen-elemen yang telah dicari recipe-nya. Selain itu, digunakan juga konsep multithreading agar dapat diterapkan paralelisme dalam pembangunan pohon solusinya.

# BAB V : Kesimpulan, Saran, dan Refleksi

## 5.1 Kesimpulan

Website ini berhasil dikembangkan menggunakan ReactJS (Vite) pada sisi frontend dan Golang pada sisi backend, dengan tujuan utama untuk membantu pengguna menemukan tree solusi resep dari elemen-elemen dalam game. Sistem mendukung dua metode pencarian, yaitu single recipe dan multi recipe, serta memanfaatkan algoritma pencarian graf: Depth-First Search (DFS) dan Breadth-First Search (BFS).

Pada sisi backend, algoritma DFS diimplementasikan secara rekursif. Pencarian dilakukan dengan menjelajahi satu jalur sedalam mungkin sebelum kembali dan mencoba jalur lainnya. DFS ini efektif dalam mencari solusi pada Tree yang dalam tapi tidak terlalu banyak resepnya/lebar. Algoritma ini juga dapat mengurangi eksplorasi simpul jika solusi ditemukan lebih awal. Sementara itu, BFS bekerja dengan menjelajahi tree secara luas terlebih dahulu melalui simpul-simpul pada level yang sama sebelum naik ke level berikutnya. BFS cocok digunakan untuk mencari jalur terpendek. Meskipun lebih sistematis, BFS cenderung membutuhkan memori lebih besar terutama untuk graf yang sangat luas.

Backend yang dibangun dengan Golang juga memanfaatkan keunggulan multithreading (melalui goroutine). Hal ini memberikan performa tinggi, terutama dalam mode multi recipe, karena banyak tree resep dapat dibentuk secara paralel tanpa saling menghambat. Hasilnya, proses pencarian menjadi jauh lebih cepat dan efisien, bahkan untuk permintaan dengan kompleksitas tinggi. Backend mengembalikan hasil berupa list tree resep unik ke frontend, lengkap dengan waktu eksekusi dan jumlah node yang dikunjungi, sehingga pengguna dapat mengetahui performa dari masing-masing metode. Memoisasi dilakukan untuk memaksimalkan hasil pencarian dan mempercepat proses eksekusi selanjutnya.

## 5.2 Saran :v

### 1. Pengujian dan Evaluasi Lebih Mendalam

Diperlukan pengujian lebih lanjut untuk mengevaluasi akurasi hasil, efisiensi algoritma, dan stabilitas sistem dalam kondisi ekstrem. Pengujian unit, integrasi, serta uji performa dapat membantu menjaga kualitas aplikasi.

### 2. Peningkatan Antarmuka Pengguna (UI/UX)

Meski fungsionalitas utama telah berjalan dengan baik, antarmuka pengguna dapat terus disempurnakan agar lebih intuitif dan mudah digunakan oleh berbagai kalangan. Desain

responsif dan visualisasi yang lebih mengutamakan user experience masih bisa dikembangkan dalam Pop up Tree

### **5.3 Refleksi**

Pengembangan website ini menjadi pengalaman yang berharga dalam memahami penerapan algoritma graf di dunia nyata. Selama proses pembangunan, kami menyadari bahwa meskipun teori algoritma seperti BFS dan DFS sudah dipahami secara konseptual, penerapannya dalam konteks aplikasi web memerlukan penyesuaian dan pengambilan keputusan teknis, seperti pemilihan struktur data yang efisien, penanganan concurrency, hingga komunikasi frontend-backend.

Secara keseluruhan, proyek ini memperluas wawasan kami tidak hanya dalam pemrograman dan algoritma, tetapi juga dalam praktik pengembangan perangkat lunak secara menyeluruh, mulai dari perancangan sistem, pemilihan teknologi, evaluasi performa dan pengalaman pengguna, hingga *deployment* dari website.



# Lampiran

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.	✓	

~ Pranala Github : [https://github.com/guntarahmbl/Tubes2\\_APAAJAUDAH.git](https://github.com/guntarahmbl/Tubes2_APAAJAUDAH.git)

~ Pranala Web : <https://tubes2-apaajaudah.vercel.app>

~ Pranala YouTube : <https://youtu.be/KCirBsWO4Gg>

# Referensi

[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)  
[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)