

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

## **Semester II Tahun Akademik 2024/2025**

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



*Disusun oleh:*

Dzubyan Ilman R. 10122010

M. Fariz Rifqi R. 13523069

Nayaka Ghana  
Subrata 13523090

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

**2025**

## DAFTAR ISI

<b>DAFTAR ISI.....</b>	<b>2</b>
<b>DAFTAR GAMBAR.....</b>	<b>4</b>
<b>DAFTAR TABEL.....</b>	<b>5</b>
<b>BAB I</b>	
<b>DESKRIPSI TUGAS.....</b>	<b>6</b>
1.1 Deskripsi Tugas.....	6
<b>BAB II</b>	
<b>LANDASAN TEORI.....</b>	<b>8</b>
2.1. Graf.....	8
2.2 Penjelajahan Graf dan Pohon.....	8
2.2.1 BFS.....	8
2.2.2 DFS.....	9
2.2.3 Bidirectional.....	10
2.2.4. Perbandingan Sifat dari Metode-Metode Penelusuran Graf.....	11
2.3 Penjelasan Singkat Aplikasi.....	11
2.4 Multithreading.....	12
2.5 Docker dan Production.....	12
<b>BAB III</b>	
<b>ANALISIS PEMECAHAN MASALAH.....</b>	<b>13</b>
3.1 Langkah-Langkah Pemecahan Masalah.....	13
3.2 Proses Pemetaan Masalah.....	17
3.2.1. Proses Pencarian Resep.....	18
3.2.1.1. Peletakan dan Visualisasi Tree.....	21
3.3 Fitur Fungsional dan Arsitektur Aplikasi.....	22
3.4 Contoh Ilustrasi Kasus.....	23
<b>BAB IV</b>	
<b>IMPLEMENTASI DAN PENGUJIAN.....</b>	<b>26</b>
4.1. Spesifikasi Teknis Program.....	26
4.1.1 Struktur Data.....	26
4.1.2 Fungsi.....	26
4.1.3 Prosedur.....	27
4.1.4 Algoritma.....	28
4.2. Tata Cara Penggunaan Program.....	34
4.2.1 Mengakses menu visual.....	35
4.2.2. Melihat visualisasi resep suatu elemen.....	35
4.2.3 Melihat performa pencarian.....	37
4.2.4 Cara Menjalankan Program (tanpa docker).....	37
4.2.5 Cara Menjalankan Program (dengan docker).....	38
4.3. Hasil Uji.....	38
4.3.1. Elemen Brick.....	38
4.3.2 Elemen Battery.....	41
4.3.3. Elemen Picnic.....	43

4.4 Analisis Hasil Uji.....	45
<b>BAB V</b>	
<b>PENUTUP.....</b>	<b>46</b>
5.1 Kesimpulan.....	46
5.2 Saran.....	46
5.3 Refleksi.....	47
<b>LAMPIRAN.....</b>	<b>49</b>
<b>DAFTAR PUSTAKA.....</b>	<b>51</b>
<b>AKHIR KATA.....</b>	<b>52</b>

## **DAFTAR GAMBAR**

Gambar 1.1 Little Alchemy 2.....	6
Gambar 1.2 Elemen dasar pada Little Alchemy 2.....	7
Gambar 2.1 Ilustrasi metode pencarian BFS.....	9
Gambar 2.2 Ilustrasi metode pencarian DFS.....	10
Gambar 3.1 Proses Aplikasi Berjalan.....	18
Gambar 3.2. Contoh Hasil Penggunaan Aplikasi.....	19
Gambar 4.1 Menu Utama.....	34
Gambar 4.2 Fitur-fitur di menu visual.....	35

## **DAFTAR TABEL**

Tabel 2.1 : Perbandingan metode BFS, DFS, dan Bidirectional berdasarkan karakteristiknya.....	11
Tabel 4.1 Output untuk elemen Brick.....	39
Tabel 4.2 Output untuk elemen Battery.....	41
Tabel 4.3 Output untuk elemen Picnic.....	43
Tabel 6.1: Poin yang dikerjakan.....	49

# BAB I

## DESKRIPSI TUGAS

### 1.1 Deskripsi Tugas



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

Little Alchemy 2 merupakan permainan berbasis web / aplikasi yang dikembangkan oleh Recloak yang dirilis pada tahun 2017, permainan ini bertujuan untuk membuat 720 elemen dari 4 elemen dasar yang tersedia yaitu air, earth, fire, dan water. Permainan ini merupakan sekuel dari permainan sebelumnya yakni Little Alchemy 1 yang dirilis tahun 2010.

Mekanisme dari permainan ini adalah pemain dapat menggabungkan kedua elemen dengan melakukan drag and drop, jika kombinasi kedua elemen valid, akan memunculkan elemen baru, jika kombinasi tidak valid maka tidak akan terjadi apa-apa. Permainan ini tersedia di web browser, Android atau iOS.

Pada Tugas Besar pertama Strategi Algoritma ini, mahasiswa diminta untuk menyelesaikan permainan Little Alchemy 2 ini dengan menggunakan **strategi Depth First Search dan Breadth First Search**.

Komponen-komponen dari permainan ini antara lain:

1. Elemen dasar

Dalam permainan Little Alchemy 2, terdapat 4 elemen dasar yang tersedia yaitu water, fire, earth, dan air, 4 elemen dasar tersebut nanti akan di-combine menjadi elemen turunan yang berjumlah 720 elemen.



**Gambar 1.2** Elemen dasar pada Little Alchemy 2  
(Sumber: <https://littlealchemy2.com/>)

## 2. Elemen turunan

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

## 3. *Combine Mechanism*

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

## **BAB II**

### **LANDASAN TEORI**

#### **2.1. Graf**

Salah satu abstraksi struktur data yang digunakan dalam penyelesaian masalah algoritmik adalah dengan graf. Tidak seperti *list* atau *array* pada umumnya, graf dapat menyajikan hubungan antara data-data yang tidak bersifat linear, sehingga penggunaannya sering diimplementasikan dalam masalah seperti *Travelling Salesman Problem*, penyelesaian Sudoku, dan lain-lain. Graf terdiri dari beberapa simpul (*node*) dan sisi-sisi (*edge*) yang menghubungkan 2 simpul. Sisi bisa merepresentasikan jalur satu arah (*directed graph*) atau jalur dua arah (*undirected graph*). Kita selalu bisa membuat sebuah lintasan (*trail*) yang mengikuti arah sisi-sisi tersebut tanpa memakai sisi yang sama dari satu simpul ke simpul lain pada sebuah graf yang terhubung (*connected graph*).

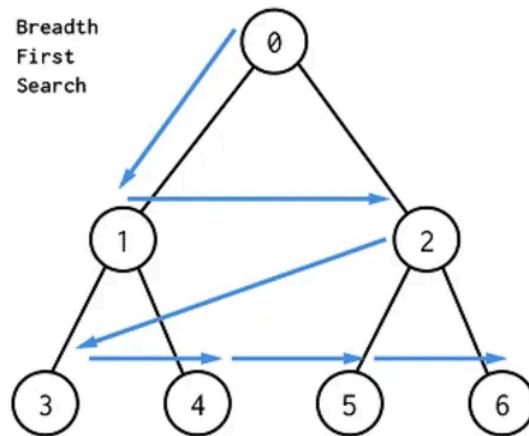
#### **2.2 Penjelajahan Graf dan Pohon**

Pohon (tree) adalah jenis graf yang terhubung sehingga kita tidak dapat menemukan siklus (*cycle*), yaitu sebuah lintasan trail yang dimulai dan berakhir di simpul yang sama. Karena bentuk struktur ini, pohon sering dipakai untuk menyajikan ruang status dari sebuah masalah dengan cabang-cabang semua kemungkinan dari status suatu objek. Sebuah simpul yang merepresentasikan keadaan awal biasanya ditandai sebagai akar (*root*) dari pohon tersebut. Saat menggunakan pohon sebagai ruang status, kita memerlukan strategi untuk menelusuri simpul-simpul dari pohon tersebut untuk mencari sebuah simpul yang menyimpan suatu konfigurasi dari objek yang kita inginkan. Penelusuran graf bisa bersifat *uninformed* dan *informed* tergantung informasi yang tersedia saat pencarian. Namun, dalam pencarian solusi, kita akan menggunakan *uninformed search*. Terdapat beberapa algoritma untuk melakukan penelusuran pohon, namun dua algoritma tipikal yang sering digunakan adalah *Breadth First Search* (BFS), *Depth First Search* (DFS), dan *Bidirectional Search*.

##### **2.2.1 BFS**

BFS adalah penelusuran simpul dalam sebuah pohon yang terlebih dahulu melihat semua simpul yang berjarak sama (diilustrasikan pada Gambar 3). Pada sebuah simpul, anak-anak dari simpul tersebut serta urutan simpul-simpul yang akan

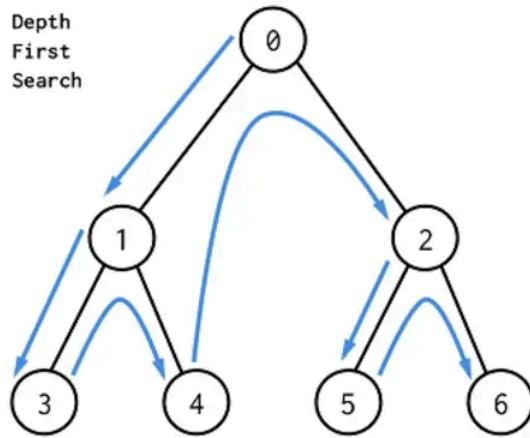
dieksplor disimpan dalam sebuah *queue*. BFS pada umumnya sangat tidak efisien dalam penggunaan memori sebab metode ini menyimpan semua cabang pada level yang sama dan pohon ruang status pada umumnya akan memiliki lebih banyak kedalaman daripada banyak cabang per simpul. Akan tetapi, BFS menjamin pemerolehan solusi optimal dari suatu masalah karena BFS mengecek level per level dari sebuah pohon status.



**Gambar 2.1** Ilustrasi metode pencarian BFS  
(sumber : [Medium](#))

### 2.2.2 DFS

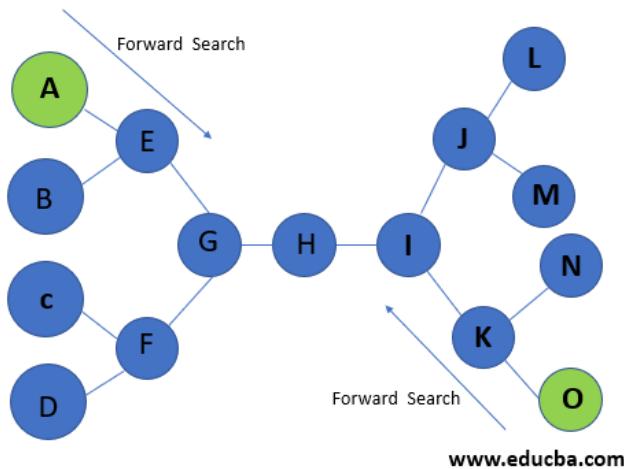
DFS adalah penelusuran simpul dalam sebuah pohon yang terlebih dahulu melihat semua simpul pada sebuah cabang yang sama sampai ke sebuah daun. Jika suatu cabang tidak memiliki anak cabang yang dapat dieksplor, kita akan melakukan backtrack ke cabang selanjutnya (diilustrasikan pada Gambar 4). Anak-anak dari simpul tersebut serta urutan simpul-simpul yang akan dieksplor disimpan dalam sebuah *stack*. Teknik ini lebih efisien dalam penggunaan memorinya karena DFS hanya menyimpan sebuah cabang tiap waktu. Namun, metode ini memiliki kelemahan yaitu, DFS dapat tidak menemukan solusi optimal secara langsung serta dapat terjebak dalam cabang yang panjang. Oleh karena itu, pemakaiannya harus diperhatikan tergantung masalah yang dimiliki.



**Gambar 2.2** Ilustrasi metode pencarian DFS  
 (sumber : [Medium](#))

### 2.2.3 Bidirectional

Bidirectional Search adalah penelusuran simpul dalam sebuah pohon yang mencari lintasan antara akar dan simpul solusi yang dimulai dari kedua simpul tersebut. Secara bergantian, kita mengeksplor simpul-simpul tetangganya sampai terdapat sebuah simpul yang telah dieksplor pada kedua pencarian tersebut. Teknik ini sangat menghemat kompleksitas waktu dibandingkan BFS dan DFS sebab pencarian hanya perlu dilakukan sampai setengah dari kedalaman pohon tersebut.



**Gambar 2.3** Ilustrasi metode pencarian Bidirectional  
 (sumber : [educba.com](#))

#### 2.2.4. Perbandingan Sifat dari Metode-Metode Penelusuran Graf

Misalkan suatu pohon ruang status memiliki maksimal  $b$  cabang pada tiap simpulnya dan  $d$  adalah kedalaman pohon tersebut. Tabel 1 di bawah ini menunjukkan karakteristik dari metode-metode penelusuran di atas.

Tabel 2.1 : Perbandingan metode BFS, DFS, dan Bidirectional berdasarkan karakteristiknya

Algoritma	Kompleksitas Ruang	Kompleksitas Waktu	Complete?	Solusi optimal dapat dicari?
BFS	$O(b^d)$	$O(b^d)$	Ya	Ya
DFS	$O(d)$	$O(d)$	Tidak, jika $d$ tak hingga	Tidak
Bidirectional	$O(b^{d/2})$	$O(b^{d/2})$	Ya	Ya

### 2.3 Penjelasan Singkat Aplikasi

Aplikasi yang dibangun adalah sebuah aplikasi yang berbentuk web. Web ini merupakan pencari resep dari sebuah elemen dari game Little Alchemy 2. Pengguna dapat mencari elemen yang diinginkan dan aplikasi ini akan menyajikan resep elemen tersebut dengan visual menarik dan intuitif. Aplikasi ini juga menyediakan pilihan metode pencarian (BFS, DFS, atau Bidirectional), opsi pencarian dengan pilihan “Resep Terpendek” atau “Beberapa Resep”, serta banyaknya resep yang ingin ditampilkan jika opsi “Beberapa Resep” terpilih. Aplikasi juga akan menampilkan banyak simpul dan waktu yang dibutuhkan untuk mencari suatu resep.

Aplikasi ini dibangun oleh dua bagian, yakni *frontend* dan *backend*. *Frontend* dibentuk dengan menggunakan bahasa Javascript, dengan *framework* Next.js dan React, sedangkan *backend* dibentuk dengan menggunakan bahasa Golang dengan *framework* Gin. Pada aplikasi ini, *frontend* berfungsi sebagai *interface* yang akan ditampilkan kepada pengguna, dan *backend* sebagai API dari aplikasi.

## 2.4 Multithreading

Dalam mencari resep yang banyak (*multiple recipe*), aplikasi ini menggunakan *multithread*, atau dalam bahasa Golang disebut sebagai *worker*. *Multithreading* adalah fitur dalam sistem operasi yang memungkinkan program melakukan beberapa tugas sekaligus. Tetapi, *multithreading* juga harus *dihandle* agar tidak menyebabkan *race condition*, sehingga diperlukan suatu *lock* atau *mutex* untuk mencegah adanya *race condition*. Dengan *multithreading*, aplikasi bisa mencari banyak resep secara cepat dengan akurasi yang cukup akurat, sehingga pengguna tidak perlu menunggu lama untuk mencari suatu resep.

## 2.5 Docker dan Production

Untuk memudahkan tahap *development* dan *production*, dibentuk Dockerfile pada masing-masing folder *frontend* dan *backend*, yang akan dipanggil oleh docker compose yang ada di root directory. Docker adalah layanan yang menyediakan kemampuan untuk mengemas dan menjalankan sebuah aplikasi dalam sebuah lingkungan terisolasi yang disebut dengan container. Dengan docker, aplikasi dapat di-*deploy* dengan lebih mudah, dan juga lebih mudah untuk melakukan *maintenance* jika aplikasi memiliki suatu *bug*, *error*, atau sekedar hanya ingin melakukan eskalasi pada aplikasi.

Pada tahap *production*, aplikasi ini menggunakan layanan Azure oleh Microsoft untuk melakukan *deploy* aplikasi. Aplikasi di-*deploy* pada sebuah *virtual machine*, yakni representasi virtual atau emulasi komputer fisik yang menggunakan perangkat lunak, bukan perangkat keras, untuk menjalankan program dan menerapkan aplikasi. Untuk akses pada public ip, hanya dibuka port 80 (port HTTP), dan juga melakukan *binding* ip ini pada sebuah domain yang bernama [seleksiasistenlabpro.xyz](http://seleksiasistenlabpro.xyz). *Binding* dilakukan dengan cara mencatat A record dari public ip vm pada registrar (aplikasi ini menggunakan registrar bernama Hostinger), sehingga aplikasi bisa diakses melalui internet.

## BAB III

### ANALISIS PEMECAHAN MASALAH

#### 3.1 Langkah-Langkah Pemecahan Masalah

Permasalahan dari permainan “Little Alchemy 2” yang diinginkan oleh spesifikasi adalah untuk mencari rangkaian elemen yang berhubungan dari suatu *base element* (*Air, Earth, Fire, Water*) menuju elemen target dengan jumlah resep satu (*single recipe*) atau lebih dari satu (*multiple recipes*) menggunakan algoritma *Breadth First Search* dan *Depth First Search* sebagai algoritma penjelajahan utama dan *Bidirectional Search* sebagai algoritma tambahan. Untuk menjalankan program, *frontend* dan *backend* perlu di-deploy secara terpisah pada *port* yang berbeda. Jika keduanya sudah di-deploy (bisa secara lokal atau internet), maka aplikasi dapat mulai untuk digunakan.

Pertama, pengguna akan masuk ke halaman web, jika web baru pertama kali di-deploy, maka web akan melakukan scraping terlebih dahulu pada website [https://little-alchemy.fandom.com/wiki/Elements\\_\(Little\\_Alchemy\\_2\)](https://little-alchemy.fandom.com/wiki/Elements_(Little_Alchemy_2)) untuk mendapatkan data resep dan gambar dari elemen. Setelah melakukan *scraping* untuk mendapatkan data, pengguna akan dihadapkan pada tampilan *landing page*, yang berisi informasi-informasi dari website. Pengguna juga bisa menyetel sebuah musik pada website untuk menemani pengguna dalam menggunakan website, sebagai suatu tambahan dari aspek *user experience*.

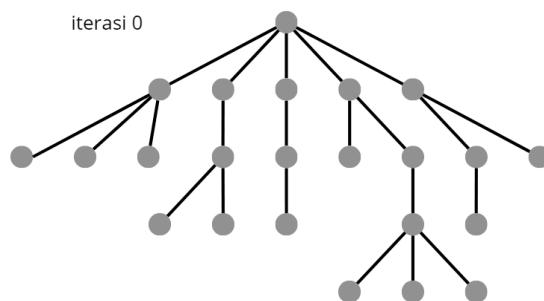
Jika pengguna melakukan scroll ke bawah, akan muncul sebuah tombol yang akan mengarahkan *endpoint* web menuju /visual, yang merupakan visualisasi dari pencarian resep terjadi. Pengguna akan dihadapkan oleh beberapa *interface*, tetapi yang utama adalah sebuah tombol berbentuk buku sebagai *list of element* yang tersedia, tombol hamburger, dan tampilan *scrollable* terminal di bawah sebagai log untuk menampilkan banyaknya simpul yang dikunjungi dan waktu eksekusi web dalam mencari resep yang diinginkan.

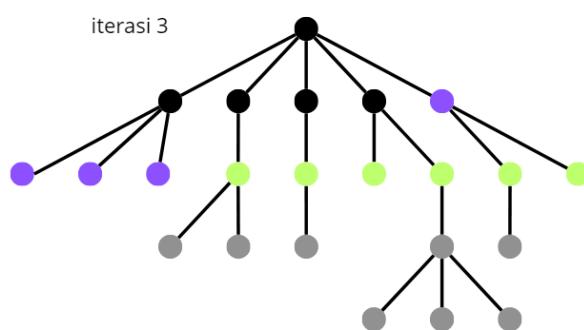
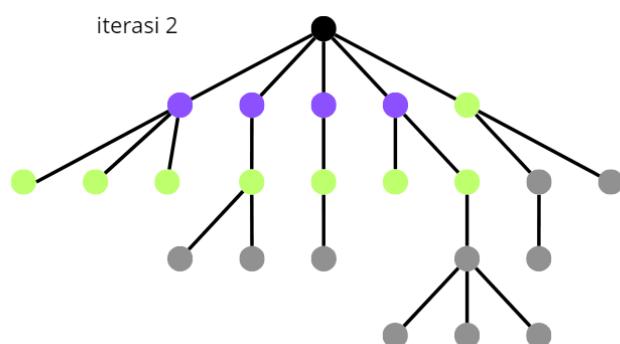
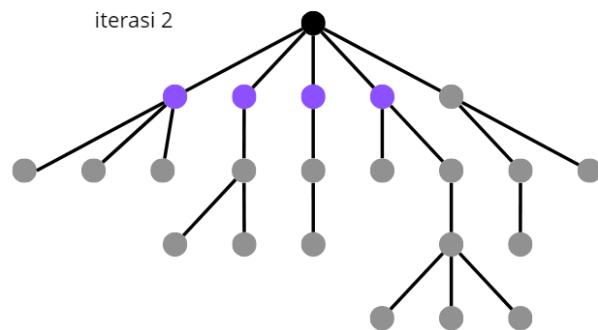
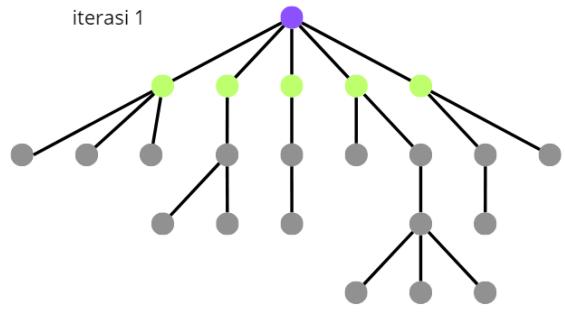
Untuk mencari suatu resep, pengguna diharuskan untuk memencet tombol hamburger, yang akan membuka suatu window baru yang berisi pilihan metode pencarian yang

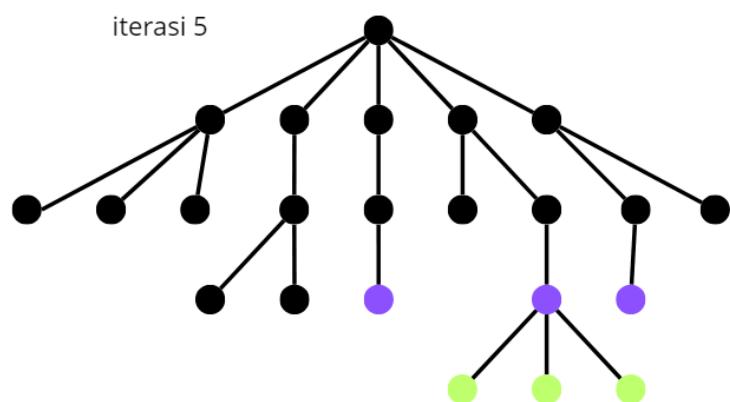
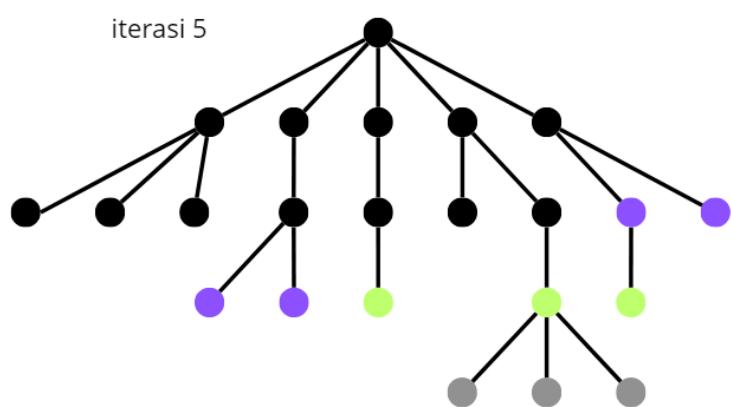
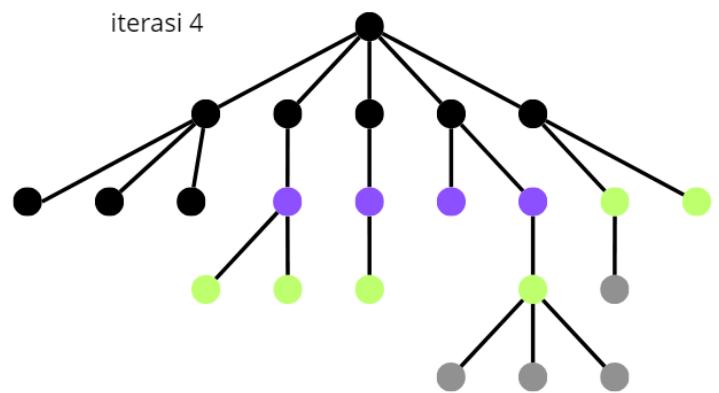
ingin digunakan (BFS, DFS, atau *Bidirectional Search*), jumlah resep yang ingin dicari (*Single* atau *Multiple recipe*), dan sebuah *search bar* untuk input elemen yang ingin dicari resepnya. Jika input dari pengguna valid, maka website akan memulai algoritma untuk mencari resepnya dan akan menampilkannya pada kanvas.

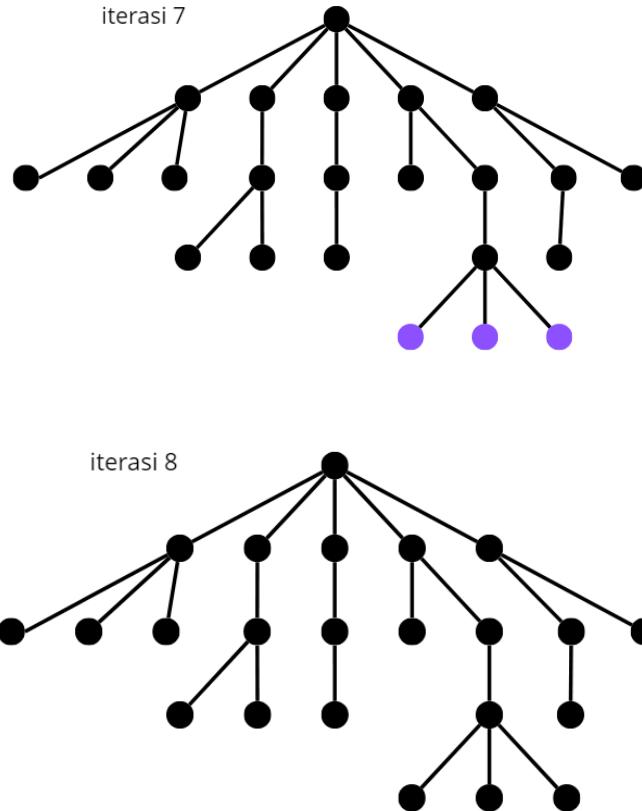
Dalam pencarian, elemen target akan menjadi simpul awal jika melihat permasalahan sebagai sebuah graf atau *root* jika melihat permasalahan sebagai sebuah pohon. Elemen-elemen lain yang membentuk elemen target akan menjadi simpul tingkat bawah dan akan membentuk suatu pohon atau graf yang lebih besar dengan semakin banyak simpul yang harus dikunjungi dan dicek. Jika pencarian berhasil dengan mengecek apakah semua *leaf* nya *base element*, maka pencarian berhasil dan hasil pencarian akan ditampilkan oleh *frontend* ke pengguna sebagai suatu pohon.

Sebagai tambahan, jika pengguna ingin mencari *multiple recipe*, maka website akan menggunakan *multithread*. *Multithreading* digunakan untuk mengoptimasi pencarian BFS maupun DFS agar lebih cepat. Selain itu, Multithreading digunakan dalam implementasi Bidirectional, karena seperti yang sudah dijelaskan pada bab sebelumnya, Bidirectional Search memerlukan dua pencari secara spontan, yakni dari source dan dari target. Multithreading adalah salah satu cara untuk mengimplementasi algoritma search tersebut, dengan membuat 1 thread untuk melakukan traversal dari source, dan 1 thread untuk melakukan traversal dari target. Di sisi lain, pada algoritma BFS dan DFS, kami menggunakan 4 thread, dimana masing-masing thread memiliki tugas yang sama, yaitu pencarian resep. Berikut ini adalah visualisasi dari penggunaan multithreading pada algoritma BFS, dimana simpul abu-abu berarti belum diproses, simpul biru artinya sedang diproses, simpul kuning artinya dalam antrian, serta simpul hitam artinya selesai diproses.



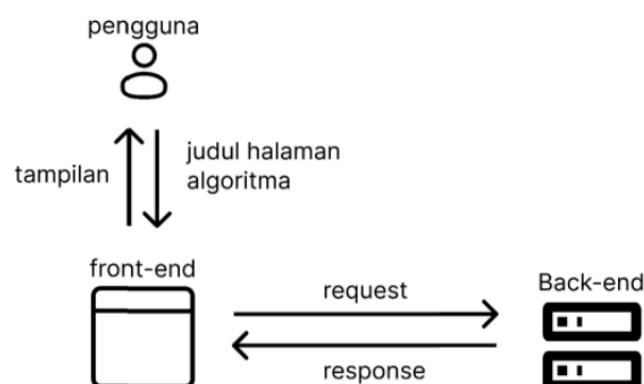






### 3.2 Proses Pemetaan Masalah

Pada dasarnya, proses penyelesaian masalah penemuan resep dapat dibagi menjadi 2 komponen utama. Setelah pengguna mengkonfigurasi preferensi pencarian elemen (elemen yang dicari, algoritma pencarian yang digunakan, dan sebagainya), *request* tersebut akan dikirimkan menuju backend. Backend kemudian akan memproses request tersebut menjadi 2 proses, yaitu proses pencarian resep dan proses visualisasi tree.



### Gambar 3.1 Proses Aplikasi Berjalan

#### 3.2.1. Proses Pencarian Resep

Proses pencarian resep pada algoritma DFS dan BFS memiliki proses dekomposisi yang sangat mirip, akan tetapi dekomposisi masalah pada algoritma *Bidirectional* sedikit berbeda. Hal tersebut akan dijelaskan lebih lanjut di paragraf selanjutnya, paragraf ini akan menjelaskan bagaimana proses dekomposisi masalah pada pencarian resep pada DFS maupun BFS. Pada tahap ini, setiap element akan dianggap sebagai suatu kelas. Selayaknya suatu kelas, bisa terdapat lebih dari 1 instansi suatu kelas. Sebagai contoh, pada gambar di bawah ini, terdapat lebih dari 1 objek “Water” yang ditampilkan.

Dalam konteks ini, setiap objek dapat direpresentasikan sebagai suatu simpul (*Vertex*), sementara hubungan resep merepresentasikan suatu garis (*Edge*), sehingga permainan ini dapat direpresentasikan sebagai suatu **Directed Graph**. Sebagai contoh, diberikan relasi bahwa : “Lava dibentuk dari menggabungkan Earth dan Fire”. Artinya, pada representasi graf, terdapat suatu simpul “Lava”, yang memiliki garis menuju simpul “Earth” ataupun “Fire”, serta perlu digaris bawahi bahwa simpul “Earth” maupun “Fire” tidak memiliki garis menuju “Lava” (Graf Berarah). Perlu diingat bahwa suatu element bisa memiliki lebih dari 1 resep. Akan tetapi, aturan yang sama tetap berlaku. Perlu digaris bawahi pula bahwa bisa saja terdapat Cycle jika menggunakan representasi resep yang diambil dari *scraping* secara utuh. Sehingga untuk menghindari adanya *infinite cycle* tersebut, kita melakukan inisialisasi untuk menentukan tier dari setiap elemen. Tier ini akan berfungsi untuk menentukan resep mana yang boleh dipilih, serta resep mana yang tidak boleh dipilih untuk menghindari adanya permasalahan *infinite cycle*.



**Gambar 3.2. Contoh Hasil Penggunaan Aplikasi**

Disambung dengan proses implementasi Algoritma BFS dan DFS untuk menyelesaikan permasalahan ini. Berikut adalah penjelasan algoritma BFS dan DFS pada *single recipe* :

- Proses BFS atau DFS dimulai dengan memasukkan membuat suatu instansi kelas dari element pertama, yaitu element **target** (element yang diinginkan oleh pengguna) ke dalam suatu struktur data, yakni *queue* untuk BFS, dan *stack* untuk DFS.
- Selama struktur data tersebut belum kosong, ambil satu objek dari struktur data tersebut. Misalkan elemen yang diambil tersebut adalah elemen X
- Cari suatu resep yang membentuk elemen X tersebut, dengan syarat bahwa 2 elemen pada resep yang dipilih harus berasal dari tier yang lebih rendah daripada elemen X. Jika ada lebih dari 1 resep yang memenuhi, pilih salah satu. Jika tidak ada resep yang memenuhi, artinya elemen X adalah elemen dasar (*Base Element*)
- Dari resep yang sudah dipilih, masukkan 2 elemen tersebut ke dalam struktur data yang sesuai.
- Ulangi langkah 2 - 4 sampai struktur data kosong

Proses pencarian pada opsi *Multiple Recipe* memiliki implementasi yang sangat mirip, hanya berbeda di langkah nomor 3. Berikut adalah penjelasan algoritma BFS dan DFS pada *multiple recipe* :

- Inisialisasi variable counter (harus atomic/*mutual exclusive*) jika menggunakan *multithread*.
- Proses BFS atau DFS dimulai dengan memasukkan membuat suatu instansi kelas dari element pertama, yaitu element **target** (element yang diinginkan

oleh pengguna) ke dalam suatu struktur data, yakni *queue* untuk BFS, dan *stack* untuk DFS.

- Selama struktur data tersebut belum kosong, ambil satu objek dari struktur data tersebut. Misalkan elemen yang diambil tersebut adalah elemen X
- Cari suatu resep yang membentuk elemen X tersebut, dengan syarat bahwa 2 elemen pada resep yang dipilih harus berasal dari tier yang lebih rendah daripada elemen X.
  - Jika variabel counter masih kurang dari banyak resep yang diinginkan, maka boleh ambil lebih dari 1 resep. Untuk setiap reset, lakukan increment pada variabel counter.
  - Jika variabel counter sudah sama dengan banyak resep yang diinginkan, artinya hanya boleh ambil 1 resep yang memenuhi.
- Dari resep yang sudah dipilih, masukkan 2 elemen tersebut ke dalam struktur data yang sesuai.
- Ulangi langkah 3 - 5 sampai struktur data kosong

Di sisi lain, abstraksi dan representasi graf pada algoritma bidirectional dapat dikatakan cukup berbeda dengan abstraksi pada algoritma DFS dan BFS. Hal tersebut dikarenakan syarat penggunaan algoritma bidirectional adalah dibutuhkannya 2 *vertex* utama, yaitu *vertex source* dan *vertex goal*. Untuk *vertex goal* sudah dapat dipastikan, yaitu *vertex* yang ingin dicari resepnya oleh pengguna. Akan tetapi bagaimana dengan *vertex source*? *Vertex source* dapat berupa base element, yaitu element-element dasar yang merupakan elemen dengan tier paling rendah. Akan tetapi, jika menggunakan abstraksi pada algoritma BFS maupun DFS, **suatu kelas (elemen) dapat diinstansikan lebih dari 1 kali**. Sebagai contoh, pada gambar 3.3, pada gambar tersebut terdapat 10 objek “Water”, 1 objek “Fire”, dan 1 objek “Earth”. Dimana ketiga elemen tersebut adalah base elemen yang pada implementasinya ketiga objek tersebut memiliki kedalaman yang berbeda-beda. Dengan adanya permasalahan ini, representasi graf yang digunakan pada metode BFS maupun DFS tidak dapat digunakan pada kasus ini. Dengan demikian, kami memutuskan untuk menggunakan pendekatan yang cukup berbeda, yakni pendekat “*Single Instantiation*”, dimana pada pendekatan ini, setiap elemen hanya boleh diinstansiasi maksimal 1 kali. Dengan menggunakan pendekatan ini, kita dapat menentukan bahwa *source vertex* adalah base element (Earth, Fire, Water, Air, and Time), sementara *goal vertex* adalah element

yang ingin dicari oleh pengguna. Kemudian, bidirectional search dapat diimplementasikan dengan melakukan 2 search, *yaitu forward search, dan backward search* yang dilakukan secara spontan atau bersamaan. Berikut ini adalah algoritma implementasi dari kedua search berikut :

- Inisialisasi 2 struktur data Queue (SourceQueue dan TargetQueue), serta 2 array of boolean untuk menentukan apakah suatu node sudah dikunjungi atau belum (VisitedBySource dan VisitedByTarget)
- Buat 2 thread, 1 thread untuk melakukan BFS dari source elemen, dan 1 thread untuk melakukan BFS dari target elemen. Traversal akan dilakukan sampai 2 pencarian beririsan seluruhnya

### 3.2.1. Peletakan dan Visualisasi Tree

Setelah didapatkan informasi-informasi yang dibutuhkan, akan dilakukan proses *peletakan* node agar visualisasi tree rapi, tidak bertabrakan, dan tidak terlalu lebar. Proses visualisasi tree dilakukan dengan algoritma *Reingold-Tilford*, yaitu algoritma klasik yang digunakan untuk menghasilkan tata letak pohon yang tidak tumpang tindih, terpusat secara hierarkis, dan kompak secara horizontal.

Algoritma ini bekerja dengan prinsip **bottom-up recursive traversal**, di mana setiap node anak dihitung terlebih dahulu posisinya, lalu posisi node induk ditentukan berdasarkan posisi anak-anaknya, memastikan keseimbangan dan konsistensi struktur pohon secara keseluruhan. Implementasi algoritma ini disesuaikan agar mendukung:

- Node dengan dua anak (representasi dari dua bahan pembentuk dalam resep),
- Struktur tree yang dapat bertingkat secara dalam (deep tree),
- Penyesuaian jarak antar node agar tidak terjadi tumpang tindih dalam visualisasi
- Proses penyelarasan ulang posisi node agar elemen yang berada dalam level yang sama tetap tersusun rapi secara horizontal.

Algoritma ini bekerja dengan sangat baik pada tree yang tiap node nya hanya memiliki 2 children. Akan tetapi, pada multiple recipes search, terdapat beberapa node yang mungkin untuk memiliki lebih dari 2 children. Dalam kasus tersebut, algoritma ini tidak menjadikan peletakan yang baik, bisa saja terdapat beberapa node tree yang

tumpang tindih, sehingga perlu adanya modifikasi dan penyesuaian lebih lanjut pada algoritma tersebut. Kelompok kami memutuskan untuk melakukan perhitungan “peletakan” simpul-simpul tersebut pada backend, dengan alasan utama **load balancing**. Front end akan melakukan yang mungkin lebih dari 500 gambar, sehingga untuk mengurangi beban front end, kelompok kami mengambil keputusan tersebut. Konsekuensi dari pengambilan keputusan ini adalah penggambaran line yang kaku, dikarenakan tidak menggunakan library yang sudah disediakan untuk visualisasi tree, seperti library D3 Tree yang tersedia pada react.

Untuk lebih lanjut mencari

### 3.3 Fitur Fungsional dan Arsitektur Aplikasi

#### Pencarian resep elemen dari game Little Alchemy 2

- Menggunakan keyword elemen sebagai input
- Menampilkan hasil pencarian dalam bentuk **graf pohon biner terbalik** (akar di bawah)

#### Visualisasi hasil pencarian

- Menggunakan **aset gambar elemen** dari game (didapat dari web scraping situs Fandom Little Alchemy 2)
- Visualisasi ditampilkan di atas **kanvas interaktif**
- Tersedia **menu di pojok atas kanvas** untuk membuka opsi pencarian

#### Fitur pencarian resep

- Input keyword elemen
- Pemilihan **metode pencarian**:
  - **BFS (Breadth-First Search)**
  - **DFS (Depth-First Search)**

- **Bidirectional Search**
- Pemilihan jenis pencarian:
  - **Resep Terpendek:** menampilkan resep dengan langkah paling sedikit
  - **Beberapa Resep:** menampilkan beberapa resep alternatif
    - Jika memilih opsi ini, muncul **kotak input tambahan** untuk menentukan jumlah resep yang ingin ditampilkan

### Kontrol visualisasi

- Pengguna dapat mengatur **lama jeda visualisasi** (delay) untuk melihat proses pembentukan pohon resep secara bertahap
- Mendukung **dragging dan zooming** menggunakan mouse untuk eksplorasi pohon resep

### Interaksi dengan elemen pada pohon

- Saat pengguna **hover di atas elemen**, akan muncul **nama unsur-unsur komposisinya**

### Output performance dari pencarian di terminal

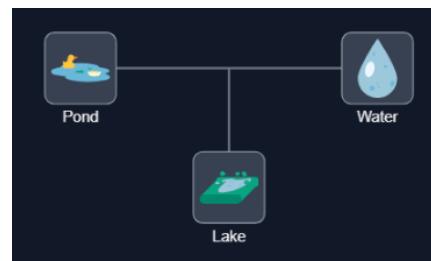
- Menampilkan **Jumlah simpul (nodes)** dari resep yang muncul
- Menampilkan **waktu pencarian** yang dibutuhkan

## 3.4 Contoh Ilustrasi Kasus

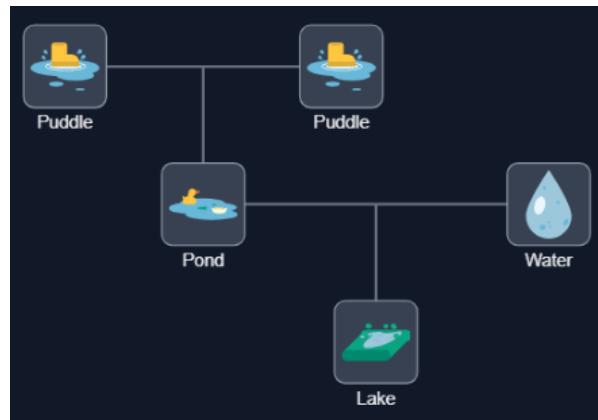
Pengguna ingin mencari resep dari “Lake”.



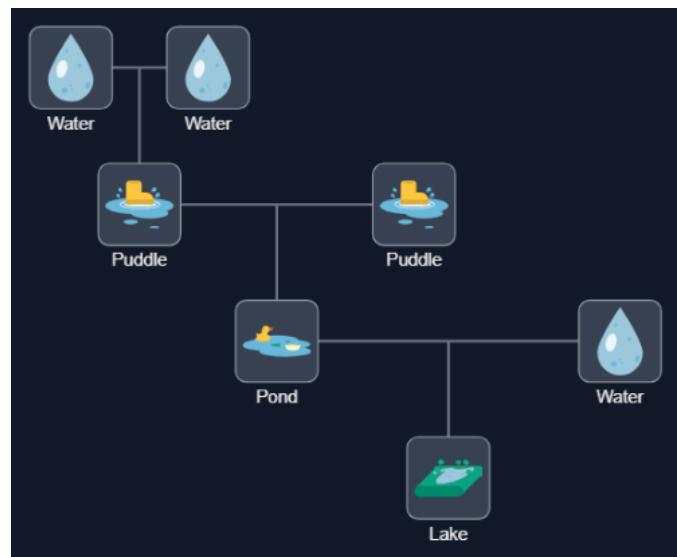
Iterasi 1 : Hanya terdapat node “Lake”



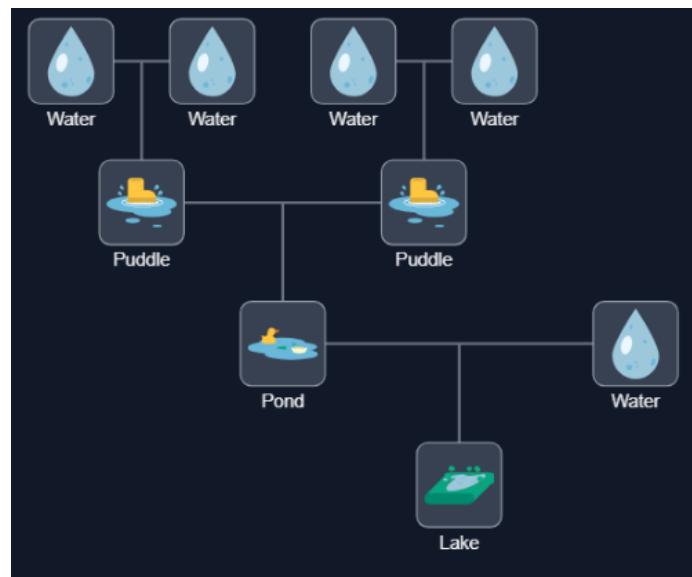
Iterasi 2 : Lake merupakan gabungan dari “Pond” dan “Water”



Iterasi 3 : “Pond” merupakan gabungan dari “Puddle” dan “Water”



Iterasi 4 : “Puddle” merupakan gabungan dari “Water” dan “Water”



Iterasi 5 : “Puddle” merupakan gabungan dari “Water” dan “Water”“

## BAB IV

### IMPLEMENTASI DAN PENGUJIAN

#### 4.1. Spesifikasi Teknis Program

##### 4.1.1 Struktur Data

Struktur data utama yang dipakai oleh aplikasi dapat dilihat di bawah ini.

1. Pair (pada file [main.go](#)), adalah struktur data sederhana yang menyimpan pasangan bahan untuk resep. Struktur ini memiliki field First dan Second yang menyimpan nama bahan pertama dan kedua.
2. Tree (pada file [main.go](#)), adalah struktur data utama untuk merepresentasikan node dalam pohon resep. Struktur ini memiliki field identitas (contohnya id, dan now untuk simpul), relasi (*parent, children, childCount*), layout (contohnya *depth, posX, posY, prelim, mod, dll*), serta thread dan ancestor.
3. ImageInfo (pada file [main.go](#)), adalah struktur yang menyimpan informasi tentang gambar elemen yang ditampilkan (contohnya *link, Row, Col, Name, Id*).
4. LineInfo (pada file [main.go](#)), adalah struktur data yang menyimpan informasi garis penghubung antar simpul, menyimpan koordinat awal dan akhir dari suatu simpul (elemen).
5. Maps (pada file [main.go](#)), adalah struktur data yang menyimpan beberapa elemen global, contohnya seperti recipes (yang menyimpan resep), nextElements (yang menyimpan tujuan elemen selanjutnya), imagesLink (yang menyimpan gambar dari elemen), dan distance (yang menyimpan jarak suatu elemen).
6. SafeTree (pada file [main.go](#)), adalah struktur data yang menggunakan mutex dan thread secara aman. Struktur data ini menyimpan queue (simpul yang akan diproses), existingTree (menyimpan reference tree yang sudah ada), dan mu (sinkronisasi dari mutex).

##### 4.1.2 Fungsi

Beberapa fungsi yang digunakan pada algoritma utama adalah sebagai berikut.

Fungsi	Deskripsi
singleDFS()	Fungsi ini melakukan pencarian resep menggunakan algoritma DFS untuk menemukan satu resep terpendek. Menghasilkan array ImageInfo dan

	LineInfo untuk visualisasi.
multiDFS()	Melakukan pencarian dengan DFS untuk menemukan beberapa resep sesuai parameter count. Dapat mencakup resep yang lebih panjang jika includeHigher true.
singleBFS()	Implementasi algoritma BFS untuk menemukan resep terpendek. Memproses node secara level-by-level dan menghasilkan data visualisasi.
multiBFS()	Versi BFS yang menemukan multiple recipes. Menggunakan atomic counter untuk membatasi jumlah resep yang ditemukan.
BidirectionalSearch()	Melakukan pencarian dua arah dari elemen dasar (Earth, Water, Air, Fire, Time) dan target secara bersamaan untuk efisiensi.
getTidyTree()	Mengimplementasikan algoritma Tidy Tree untuk menghasilkan layout pohon yang rapi dengan menghitung posisi x,y setiap node.
getImageURL()	Fungsi helper untuk menghasilkan URL gambar yang dinamis berdasarkan host request.
fetchImages()	Fungsi frontend yang melakukan request ke API backend, memproses response, dan memuat gambar secara incremental dengan delay.

#### 4.1.3 Prosedur

Beberapa prosedur yang digunakan pada algoritma utama adalah sebagai berikut.

Prosedur	Deskripsi
INITIALIZE()	Prosedur utama untuk inisialisasi aplikasi. Memeriksa keberadaan file data, menjalankan scraper jika perlu, dan memuat recipes, images, serta menghitung distances.

readRecipes()	Membaca file CSV recipes dan membangun map recipes serta nextElements. Setiap resep diparsing untuk membuat relasi antar elemen.
readImages()	Membaca file CSV images dan membangun map imagesLink yang memetakan nama elemen ke URL gambar.
findAllDistances()	Menghitung jarak setiap elemen dari elemen dasar menggunakan pendekatan iteratif. Elemen dasar memiliki distance 0.
runScraperProcess()	Menjalankan proses scraping untuk menghasilkan file data jika belum ada. Dapat menjalankan binary atau go run tergantung environment.
handleSearch()	Handler HTTP untuk endpoint /api. Memproses request, memanggil algoritma yang sesuai, dan mengembalikan response JSON.
resolveOverlaps()	Prosedur dalam tidy tree untuk memastikan tidak ada subtree yang overlap dengan melakukan shifting jika diperlukan.
normalizeCoordinates()	Menormalisasi koordinat dalam tree untuk memastikan semua posisi bernilai positif dengan shifting jika ada koordinat negatif.
centerParents()	Memastikan setiap parent node terpusat di atas children-nya untuk layout yang seimbang.

#### 4.1.4 Algoritma

Pseudocode untuk algoritma BFS

```
function BFS(input target : string) -> ImageInfo[],  
LineInfo[]  
{ melakukan proses pencarian resep menggunakan algoritma  
BFS }
```

DEKLARASI

```

tree : Tree
queue : Queue
visited : Set
images : ImageInfo[]
lines : LineInfo[]
countId : integer = 0

ALGORITMA
tree = new Tree(target)
queue.enqueue(tree)
visited.add(target)

while not queue.isEmpty() do
    node = queue.dequeue()
    node.id = countId
    countId = countId + 1

    for each recipe in recipes[node.now] do
        if max(distances[recipe.First],
distances[recipe.Second]) + 1 == distances[node.now]
then
            left = new Tree(recipe.First)
            right = new Tree(recipe.Second)
            node.children.add(left, right)

            queue.enqueue(left)
            queue.enqueue(right)
            break
        endif
    endfor
endwhile

getTidyTree(tree)
generateVisualization(tree, images, lines)
return images, lines

```

### Pseudocode untuk algoritma DFS

```

function DFS(input target : string) -> ImageInfo[],
LineInfo[]
{ melakukan proses pencarian resep menggunakan algoritma
DFS }

DEKLARASI
tree : Tree
stack : Stack
images : ImageInfo[]

```

```

lines : LineInfo[]
countId : integer = 0

ALGORITMA
tree = new Tree(target)
stack.push(tree)

while not stack.isEmpty() do
    node = stack.pop()
    node.id = countId
    countId = countId + 1

    for each recipe in recipes[node.now] do
        if max(distances[recipe.First],
distances[recipe.Second]) < distances[node.now] then
            left = new Tree(recipe.First)
            right = new Tree(recipe.Second)
            node.children.add(left, right)

            stack.push(right)
            stack.push(left)
            break
        endif
    endfor
endwhile

getTidyTree(tree)
generateVisualization(tree, images, lines)
return images, lines

```

Pseudocode untuk algoritma *Bidirectional Search*

```

function BidirectionalSearch(input target : string) ->
ImageInfo[], LineInfo[]
{ melakukan pencarian dua arah dari elemen dasar dan
target }

DEKLARASI
queueSource : Queue
queueTarget : Queue
visitedBySource : Map
visitedByTarget : Map
mapTree : Map<string, Tree>
images : ImageInfo[]
lines : LineInfo[]

ALGORITMA
// Inisialisasi queue dari sumber
for each basicElement in ["Earth", "Water", "Air",

```

```

"Fire", "Time"] do
    mapTree[basicElement] = new Tree(basicElement)
    queueSource.enqueue(mapTree[basicElement])
    visitedBySource[basicElement] = true
endfor

// Inisialisasi queue dari target
mapTree[target] = new Tree(target)
queueTarget.enqueue(mapTree[target])
visitedByTarget[target] = true

while not queueSource.isEmpty() and not
queueTarget.isEmpty() do
    // BFS dari source
    if not queueSource.isEmpty() then
        nodeSource = queueSource.dequeue()

        for each next in nextElements[nodeSource.now] do
            if visitedByTarget.contains(next) then
                // Path ditemukan
                generatePath(mapTree, images, lines)
                return images, lines
            endif

            if not visitedBySource.contains(next) then
                for each recipe in recipes[next] do
                    if visitedBySource.contains(recipe.First) and
                    visitedBySource.contains(recipe.Second)
then
                    mapTree[next] = new Tree(next)

                    mapTree[next].children.add(mapTree[recipe.First],
mapTree[recipe.Second])
                    queueSource.enqueue(mapTree[next])
                    visitedBySource[next] = true
                    break
                endif
            endif
        endfor
    endif
endif
endif

// BFS dari target
if not queueTarget.isEmpty() then
    nodeTarget = queueTarget.dequeue()

    for each recipe in recipes[nodeTarget.now] do
        if distances[recipe.First] + 1 ==
distances[nodeTarget.now] or
        distances[recipe.Second] + 1 ==

```

```

distances[nodeTarget.now] then
    if not visitedByTarget.contains(recipe.First)
then
    mapTree[recipe.First] = new Tree(recipe.First)
    queueTarget.enqueue(mapTree[recipe.First])
    visitedByTarget[recipe.First] = true
endif

    if not visitedByTarget.contains(recipe.Second)
then
    mapTree[recipe.Second] = new
Tree(recipe.Second)
    queueTarget.enqueue(mapTree[recipe.Second])
    visitedByTarget[recipe.Second] = true
endif
endif
endfor
endif
endwhile

generateVisualization(mapTree, visitOrder, images,
lines)
return images, lines

```

Pseudocode untuk algoritma *Tidy Tree* (beserta *helper*-nya)

```

function getTidyTree(input root : Tree) -> void
{ menghasilkan layout pohon yang rapi menggunakan
algoritma Tidy Tree }

DEKLARASI
maxDepth : integer

ALGORITMA
// Tahap 1: Assign depths
assignDepths(root, 0)

// Tahap 2: Calculate initial positions
calculateInitialPositions(root, 0)

// Tahap 3: Resolve overlaps
resolveOverlaps(root)

// Tahap 4: Center parents
centerParents(root)

// Tahap 5: Normalize coordinates
normalizeCoordinates(root)

```

```

// Tahap 6: Finalize positions
collectTree(root)

procedure calculateInitialPositions(input node : Tree,
input offset : integer) -> integer
{ menghitung posisi x awal untuk setiap node }

DEKLARASI
childOffset : integer
firstChild : Tree
lastChild : Tree

ALGORITMA
if node.children.isEmpty() then
    node.posX = offset
    node.posY = node.depth * LEVEL_SEP
    return offset + NODE_WIDTH
endif

childOffset = offset

for each child in node.children do
    childOffset = calculateInitialPositions(child,
childOffset)
    if child != node.children.last() then
        childOffset = childOffset + SIBLING_SEP
    endif
endfor

firstChild = node.children.first()
lastChild = node.children.last()
node.posX = (firstChild posX + lastChild posX) / 2
node.posY = node.depth * LEVEL_SEP

return childOffset

procedure resolveOverlaps(input root : Tree) -> void
{ memastikan tidak ada subtree yang overlap }

DEKLARASI
levelMap : Map<integer, Tree[]>
maxDepth : integer

ALGORITMA
collectNodesByLevel(root, levelMap)
maxDepth = getMaxDepth(levelMap)

for depth from maxDepth downto 0 do
    nodesAtLevel = levelMap[depth]
    sortByXPosition(nodesAtLevel)

```

```

for i from 1 to nodesAtLevel.length - 1 do
    leftNode = nodesAtLevel[i-1]
    rightNode = nodesAtLevel[i]

    leftExtent = getSubtreeRightExtent(leftNode)
    rightExtent = getSubtreeLeftExtent(rightNode)

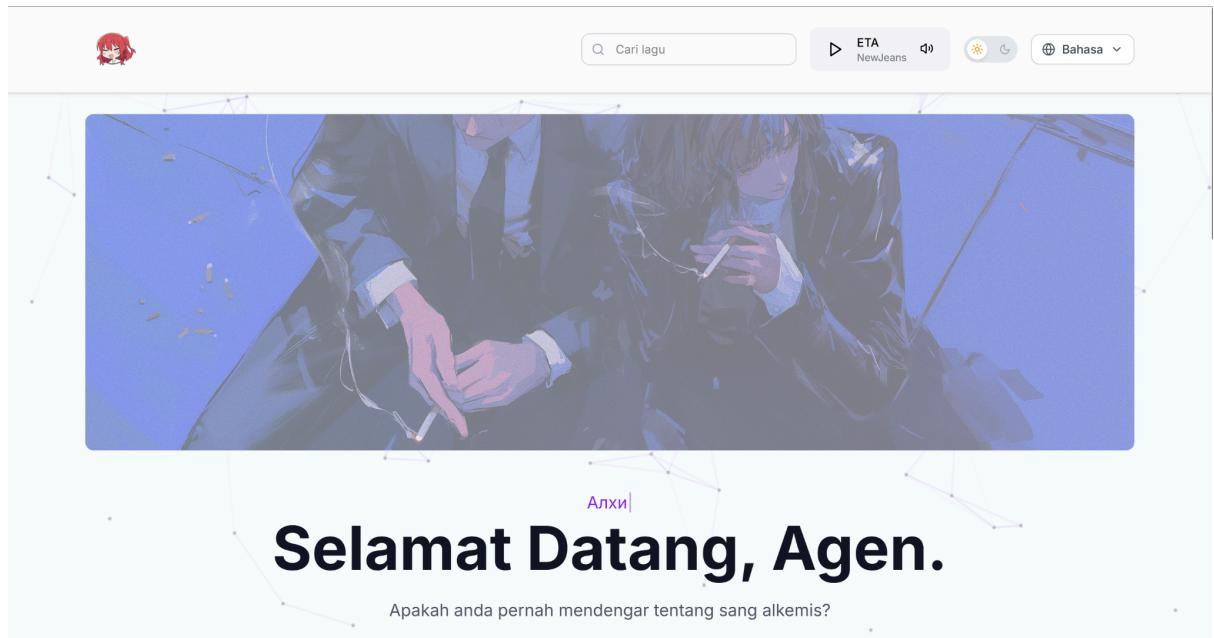
    gap = rightExtent - leftExtent

    if gap < SUBTREE_SEP then
        shift = SUBTREE_SEP - gap
        shiftSubtree(rightNode, shift)
    endif
endfor

for each node in nodesAtLevel do
    if not node.children.isEmpty() then
        centerNodeOverChildren(node)
    endif
endfor
endfor

```

## 4.2. Tata Cara Penggunaan Program



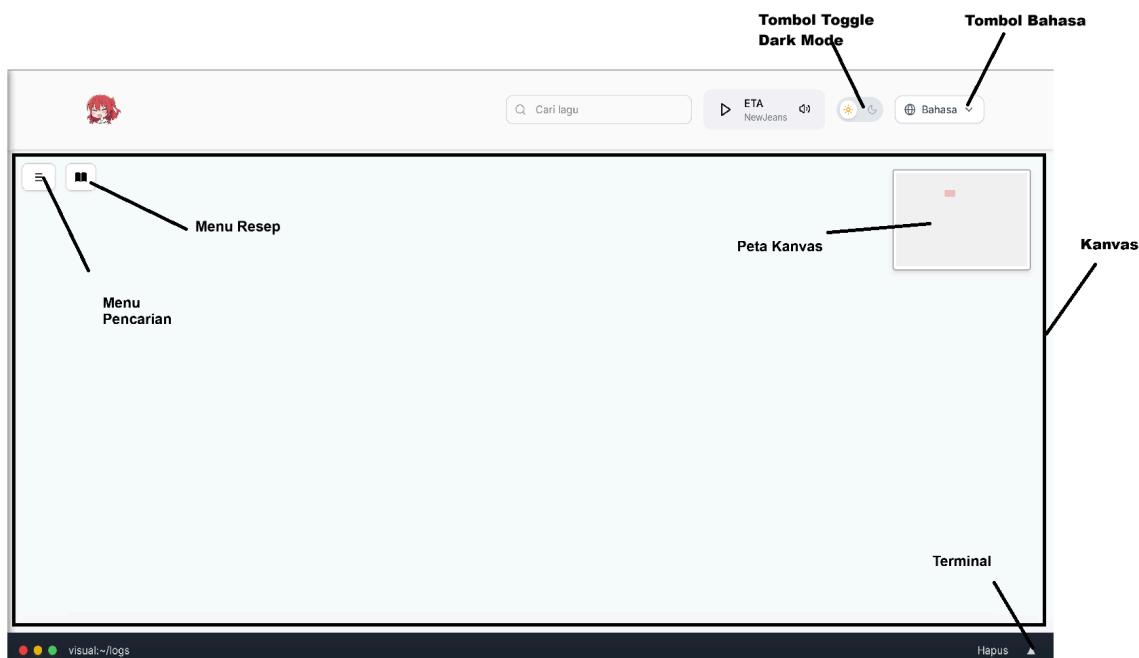
**Gambar 4.1** Menu Utama

Situs [seleksiasistenlabpro.xyz](http://seleksiasistenlabpro.xyz) terdiri dari 2 menu: menu utama (*landing page*) dan menu visual. Untuk menggunakan aplikasi, pengguna harus masuk ke menu visual.

#### 4.2.1 Mengakses menu visual

1. Masuk ke situs [seleksiasistenlabpro.xyz](http://seleksiasistenlabpro.xyz)
2. Klik tombol Mulai Perjalanan yang berada di bawah laman situs
3. Tunggu situs memuat laman sampai muncul menu seperti pada Gambar 4.2

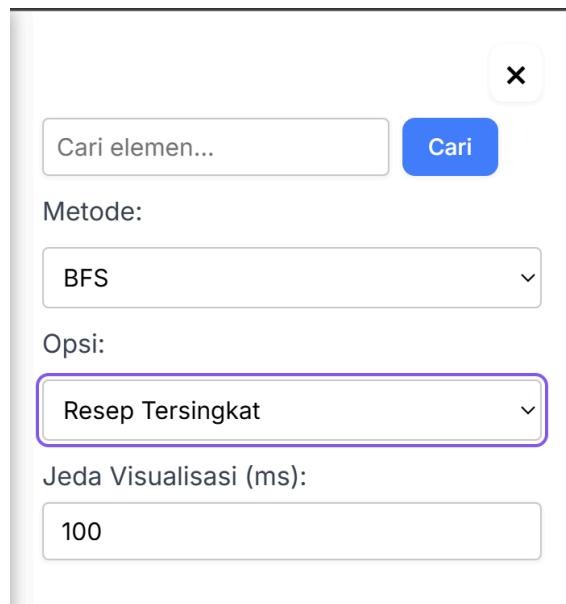
Di menu visual, pengguna akan melihat fitur-fitur aplikasi ini sebagai di gambar berikut.



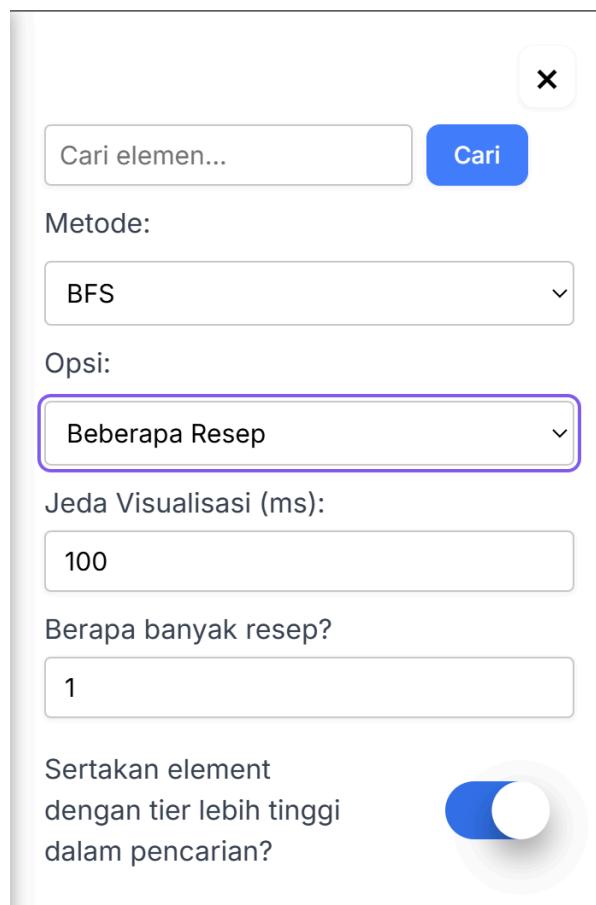
**Gambar 4.2** Fitur-fitur di menu visual

#### 4.2.2 Melihat visualisasi resep suatu elemen

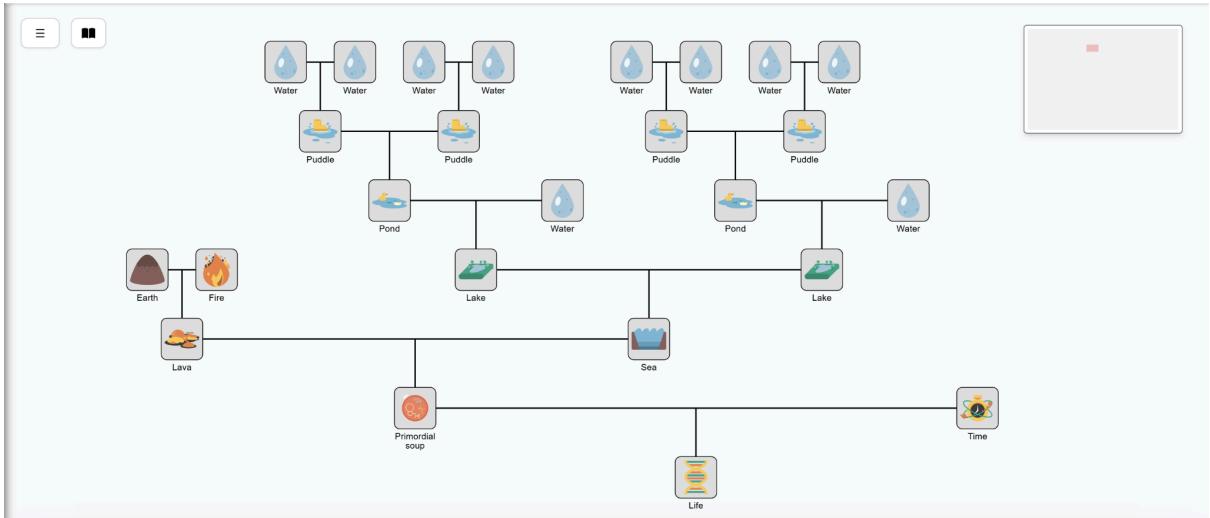
1. Tekan tombol Menu Pencarian di pojok kiri atas
2. Masukkan nama elemen yang resepnya ingin divisualisasikan ke kotak pencarian
3. Pilih metode pencarian yang ingin digunakan pada menu dropdown dengan label "Metode"
4. Pilih opsi pencarian yang diinginkan pada menu dropdown "Opsi :"
5. Masukkan lama jeda visualisasi yang diinginkan. Masukkan harus lebih dari 0.
6. Apabila terpilih opsi "Beberapa Resep", masukkan banyak resep yang diinginkan, dan tentukan apakah kita ingin menampilkan resep dengan elemen dengan tier lebih tinggi, yaitu elemen yang panjang resep terpendeknya lebih panjang daripada elemen yang kita sedang cari.
7. Tekan tombol "Cari"
8. Tunggu sampai visual ter-render dengan sempurna.



**Gambar 4.3** Menu Pencarian



**Gambar 4.4** Menu alternatif jika opsi “Beberapa Resep” terpilih



**Gambar 4.5** Contoh hasil visualisasi untuk elemen “Life” dengan metode BFS dan opsi “Shortest Recipe”

#### 4.2.3 Melihat performa pencarian

1. Tekan tombol segitiga di pojok kanan bawah.
2. Scroll ke bagian pencarian yang performanya ingin dilihat

```
[8:02:52 PM] ✓ Server berhasil merespons
[8:02:52 PM] Node yang Dikunjungi: 25
[8:02:52 PM] Waktu Eksekusi Server: 107ms
[8:03:14 PM] ✓ Visualisasi selesai
[8:03:14 PM] Waktu Permintaan Klien: 22106ms
```

**Gambar 4.6** Contoh output performa untuk pencarian resep elemen “Life” dengan metode BFS dan opsi “Shortest Recipe”

#### 4.2.4 Cara Menjalankan Program (tanpa docker)

Untuk menjalankan program, diperlukan dua terminal, terminal pertama akan menjalankan *frontend*, dan terminal kedua akan menjalankan *backend*. Aplikasi hanya dapat berjalan jika kedua bagian ini telah menyala, karena saling bergantung satu sama lain. Langkah-langkah untuk menjalankannya adalah sebagai berikut:

##### A. *Frontend*

1. Buka sebuah terminal
2. Lakukan clone pada repository (terlampir pada bagian lampiran), atau dengan melakukan command “git clone [nama repository]”
3. Masuk ke folder bernama Tubes2\_Labpro-Hebat/frontend

4. Lakukan instalasi *package* dan *dependencies* dengan cara mengetik “npm i” atau “npm install” pada terminal, tunggu hingga proses selesai,
5. Jalankan *frontend* dengan cara mengetik “npm run dev”, dan akses <http://localhost:3000> pada browser.

*B. Backend*

1. Buka sebuah terminal baru (yang berbeda dari *frontend*)
2. Jika pada bagian *frontend* belum melakukan clone, maka langkahnya sama seperti *frontend*, jika sudah, lewati langkah ini
3. Masuk ke folder bernama Tubes2\_Labpro-Hebat/backend
4. Jalankan *backend* dengan cara mengetik “go run [main.go](#)”, tunggu prosesnya hingga selesai.

**4.2.5 Cara Menjalankan Program (dengan docker)**

Untuk menjalankan program,hanya diperlukan compose dari docker. Aplikasi hanya dapat berjalan jika kedua bagian ini telah menyala, karena saling bergantung satu sama lain. Langkah-langkah untuk menjalankannya adalah sebagai berikut:

*A. Development*

1. Buka sebuah terminal
2. Lakukan clone pada repository (terlampir pada bagian lampiran), atau dengan melakukan command “git clone [nama repository]”
3. Masuk ke folder bernama Tubes2\_Labpro-Hebat
4. Up dan build *frontend* dan *backend* dengan cara mengetik “docker-compose -f docker-compose.yml up –build”, tunggu hingga proses selesai

*B. Production*

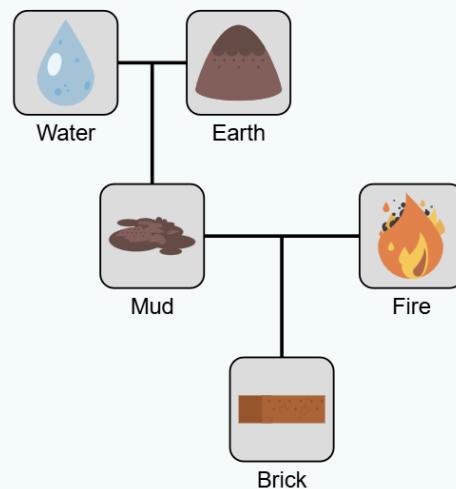
1. Buka sebuah terminal
2. Lakukan clone pada repository (terlampir pada bagian lampiran), atau dengan melakukan command “git clone [nama repository]”
3. Masuk ke folder bernama Tubes2\_Labpro-Hebat
4. Up dan build *frontend* dan *backend* dengan cara mengetik “docker-compose -f docker-compose.prod.yml up –build”, tunggu hingga proses selesai

## 4.3. Hasil Uji

### 4.3.1. Elemen Brick

Tabel 4.1 Output untuk elemen Brick

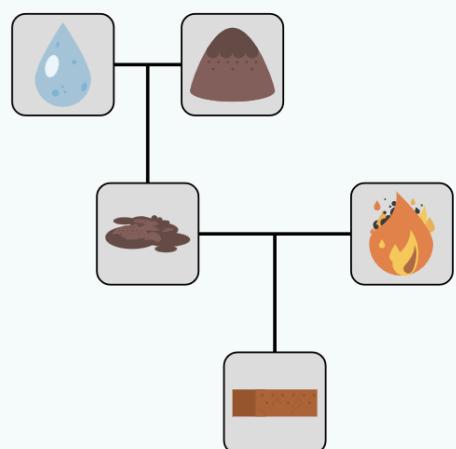
Shortest Recipe, BFS



Waktu eksekusi : 39ms

Simpul yg dikunjungi : 5

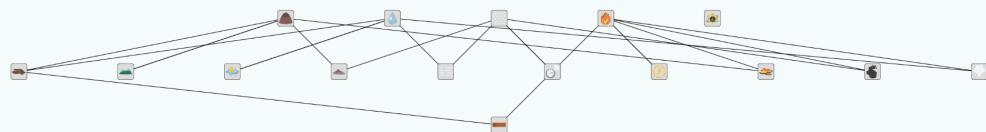
Shortest Recipe, DFS



Waktu eksekusi : 31ms

Simpul yg dikunjungi : 5

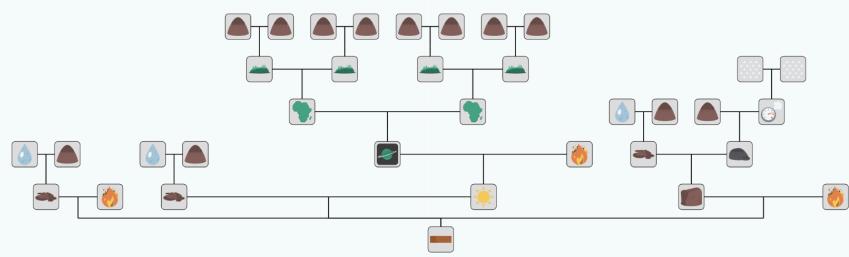
### Shortest Recipe, Bidirectional



Waktu eksekusi : 66ms

Simpul yg dikunjungi : 16

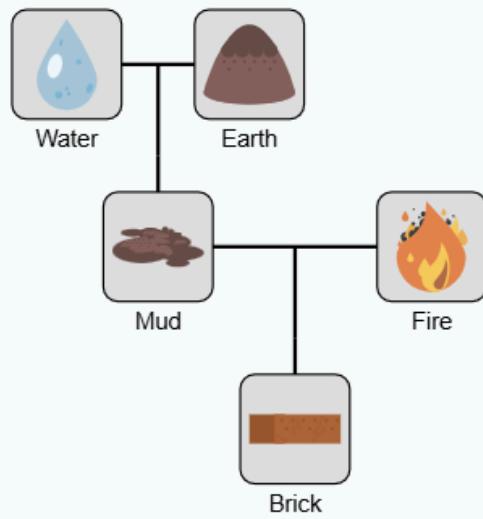
### Multi Recipe untuk 4 resep, BFS, dengan Toggle Higher Tier Element



Waktu eksekusi : 38ms

Simpul yang dikunjungi : 35

### Multi Recipe untuk 4 resep, BFS, **tanpa** Toggle Higher Tier Element



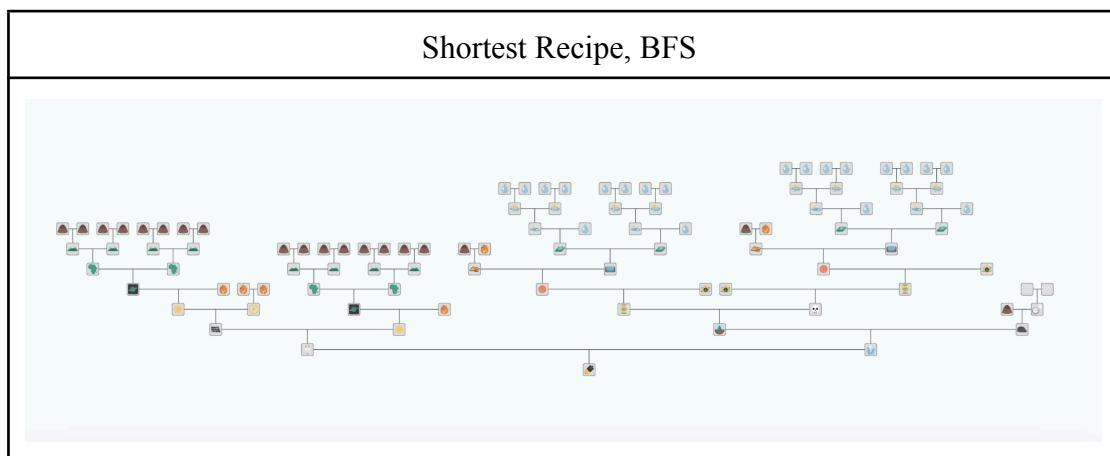
Waktu eksekusi : 36ms

Jumlah simpul yang dikunjungi : 5

Catatan : Elemen Brick hanya memiliki satu resep tanpa elemen yang tier-nya lebih besar. Jadi, walaupun kita meminta 4, hanya 1 resep yang dikeluarkan.

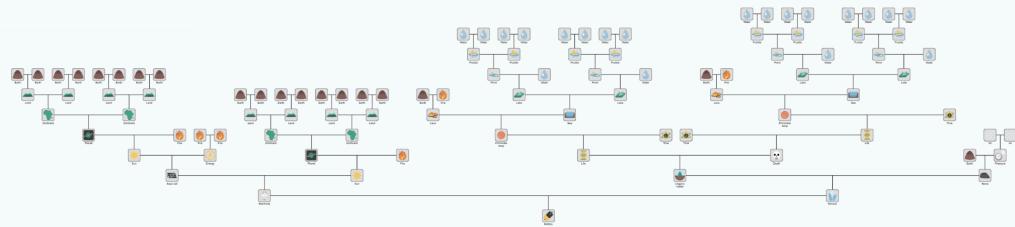
#### 4.3.2 Elemen Battery

Tabel 4.2 Output untuk elemen Battery



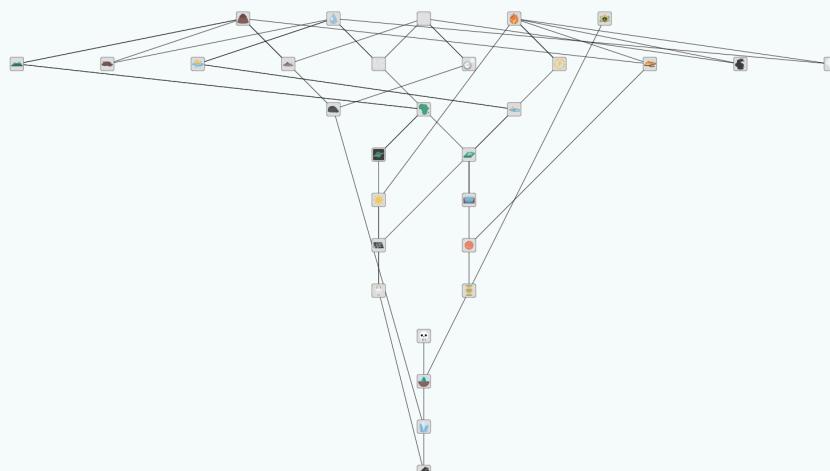
Waktu eksekusi : 52ms  
Simpul yg dikunjungi : 99

#### Shortest Recipe, DFS



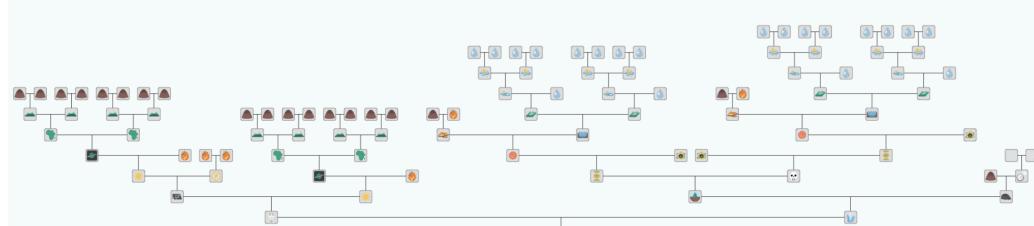
Waktu eksekusi : 58ms  
Simpul yg dikunjungi : 99

#### Shortest Recipe, Bidirectional

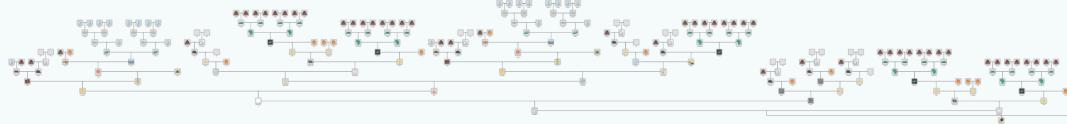


Waktu eksekusi : 42ms  
Simpul yg dikunjungi : 30

#### Multi Recipe untuk 2 resep, BFS, dengan Toggle Higher Tier Element



Cabang 1

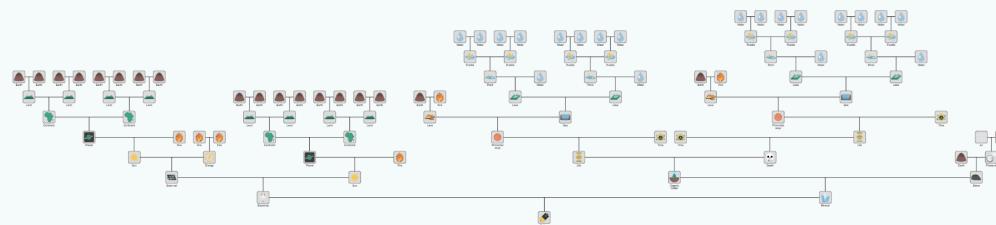


Cabang 2

Waktu eksekusi : 198ms

Simpul yg dikunjungi : 315

Multi Recipe untuk 2 resep, BFS, **tanpa** Toggle Higher Tier Element



Waktu eksekusi : 87ms

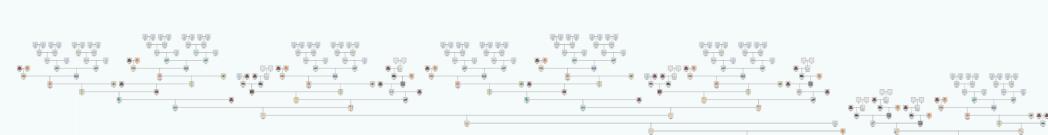
Simpul yg dikunjungi : 99

Catatan : Elemen Battery hanya memiliki satu resep tanpa elemen yang tier-nya lebih besar. Jadi, walaupun kita meminta 2, hanya 1 resep yang dikeluarkan.

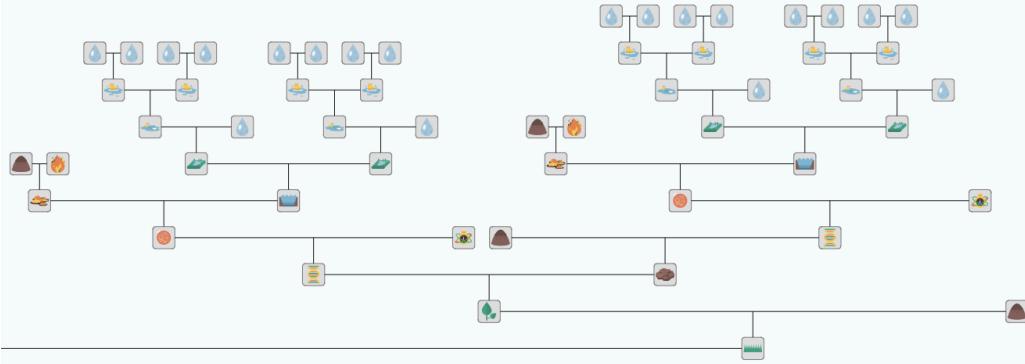
#### 4.3.3. Elemen Picnic

Tabel 4.3 Output untuk elemen Picnic

Shortest Recipe, BFS



Cabang 1

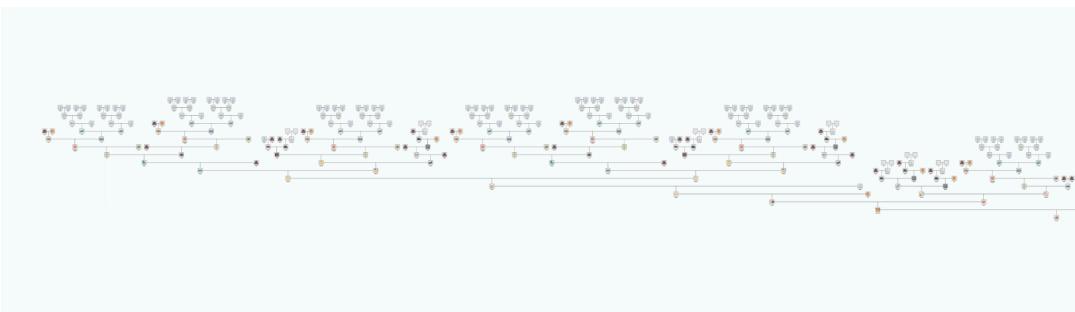


Cabang 2

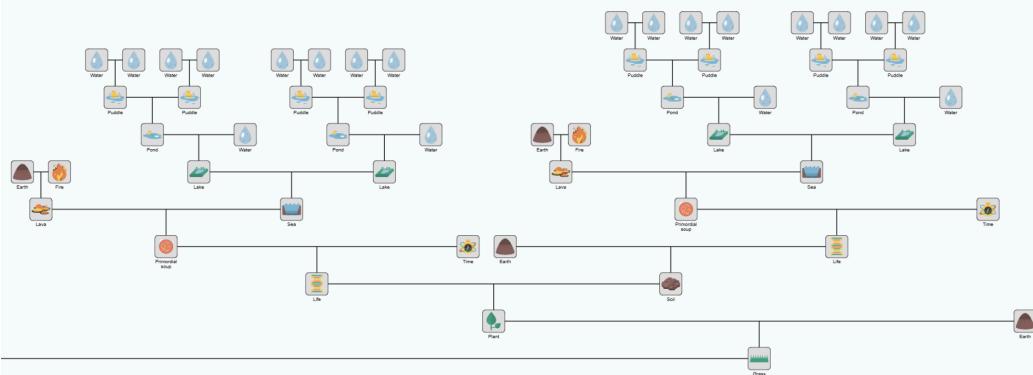
Waktu eksekusi : 112ms

Simpul yg dikunjungi : 319

### Shortest Recipe, DFS



Cabang 1

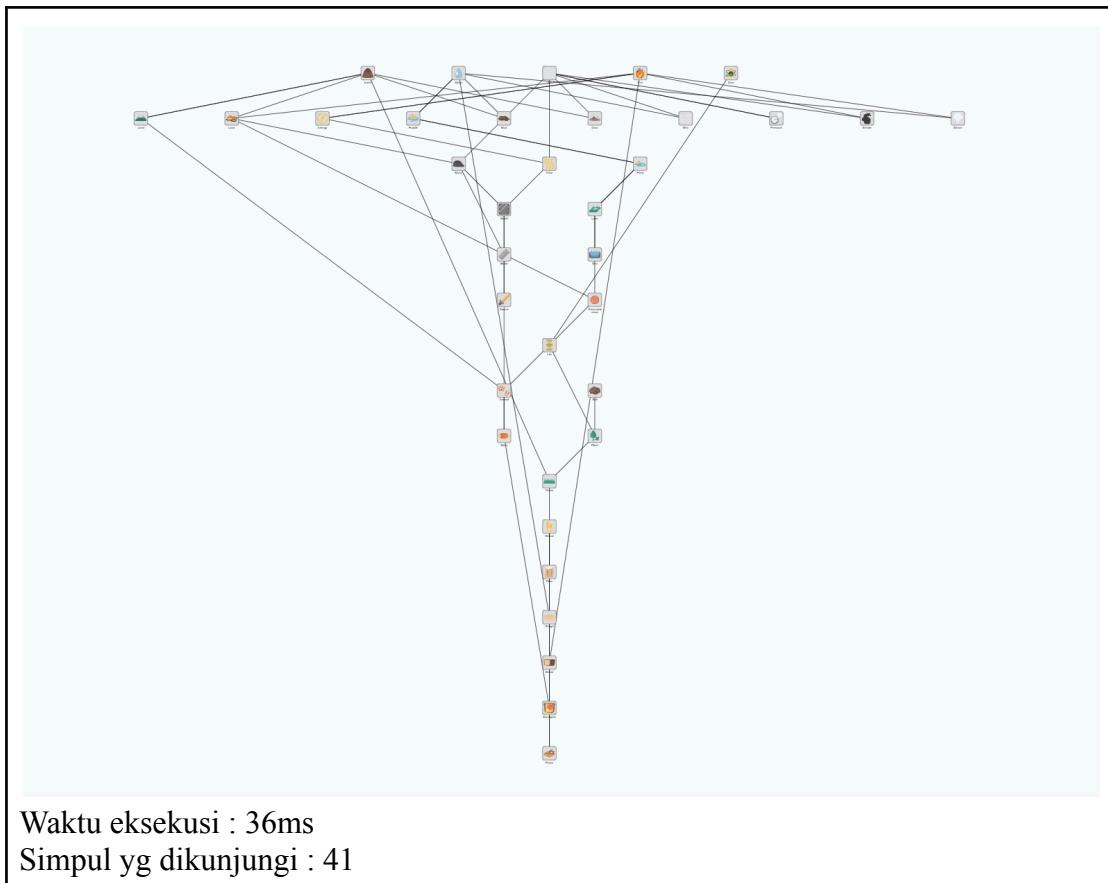


Cabang 2

Waktu eksekusi : 123ms

Simpul yg dikunjungi : 319

### Shortest Recipe, Bidirectional.



#### 4.4 Analisis Hasil Uji

Berdasarkan dari tes uji sebelumnya, dapat dilihat bahwa dari ketiga metode pencarian tersebut, metode bidirectional menggunakan waktu eksekusi yang lebih cepat dibandingkan dengan metode BFS maupun DFS untuk elemen dengan tier yang cukup tinggi seperti Battery dan Picnic, namun lebih lambat pada elemen dengan tier rendah seperti Brick. Ini sesuai dengan sifat pencarian Bidirectional yang telah dibahas pada bab 2.2.3. Sementara itu, antara metode BFS dan DFS, tidak ada perbedaan yang signifikan dari waktu eksekusi maupun simpul yang dikunjungi.

Ini dikarenakan, metode Bidirectional harus mencari lintasan dari simpul awal dan simpul akhir yang ketika jarak antara kedua simpul ini cukup besar, penghematan waktu yang diperoleh akan semakin banyak ketimbang saat jaraknya pendek.

## **BAB V**

## **PENUTUP**

### **5.1 Kesimpulan**

Pada Tugas Besar 2 IF2211-Strategi Algoritma Semester 2 Tahun Ajaran 2024/2025, kami diminta untuk membuat suatu aplikasi berbasis website yang menyelesaikan persoalan pada permainan “Little Alchemy 2” dengan menggunakan strategi penjelajahan graf: *Breadth First Search* (BFS), *Depth First Search* (DFS), dan bonus yakni *Bidirectional Search*.

Kami berhasil membuat website yang dapat mencari resep-resep yang ada di dalam permainan “Little Alchemy 2” dengan cukup baik. Untuk menyelesaikan permasalahan tersebut, kami menggunakan bahasa pemrograman Golang dengan *framework* Gin sebagai *Application Programming Interface* (API) dan juga *backend* untuk mendapatkan data pada permainan “Little Alchemy 2” dengan cara scraping dan juga melakukan pencarian resep dengan menggunakan algoritma penjelajahan graf. Selain itu, kami juga membangun *frontend* website dengan menggunakan bahasa utama Javascript dengan bantuan CSS. Bagian *frontend* juga dibangun dengan *framework* Next.js dan React, sebagai *interface* yang akan ditampilkan kepada pengguna dan juga sebagai *interface* dari hasil penjelajahan graf yang telah dilakukan.

### **5.2 Saran**

Pelaksanaan Tugas Besar IF2211-Strategi Algoritma Semester 2 Tahun Ajaran 2024/2025 merupakan pengalaman berharga dan juga berkesan bagi kami. Dari pengalaman ini, selain belajar tentang strategi dalam menjelajahi graf untuk menaklukkan persoalan pencarian, kami juga belajar bagaimana cara untuk membangun website dan juga men-deploy website tersebut hingga bisa diakses secara publik. Kami juga ingin memberikan beberapa saran kepada pembaca yang mungkin akan menghadapi tugas serupa di masa depan atau tertarik untuk mengembangkan topik ini lebih lanjut:

1. Jadwalkan dan buat *roadmap* yang jelas pada pengembangan aplikasi. Karena aplikasi yang ingin dibangun secara *scalability*-nya menengah sampai tinggi,

- maka hal yang bijak jika dibangun suatu jadwal dan *roadmap* terlebih dahulu agar proses penggerjaan lebih terarah dan juga terstruktur.
2. Komunikasi antar anggota tim. Jika pengembangan aplikasi dilakukan dengan tim, maka komunikasi antar anggota sangat penting untuk mengetahui *progress* antar anggota dan juga menghindari suatu konflik dalam pekerjaan yang sedang dikerjakan.
  3. Eksplorasi algoritma penjelajahan graf. Penjelajahan graf bisa dieksplorasi lebih lanjut dengan menggunakan algoritma *Iterative Deepening Search* (IDS), *Depth Limited Search* (DLS), atau algoritma lainnya. Penggunaan algoritma BFS, DFS, dan *Bidirectional* hanya sebagai beberapa contoh dari algoritma penjelajahan graf, dan masih bisa dikembangkan lebih lanjut.
  4. Eksplorasi bahasa pemrograman dan *framework*. Bahasa pemrograman dan *framework* yang digunakan bisa dieksplorasi lebih lanjut, tidak hanya sebatas Golang dan Javascript dengan menggunakan *framework* Gin, Next.js, dan React. Bahasa dan *framework* lain bisa digunakan agar aplikasi yang dikembangkan lebih optimal. Sebagai contoh, penggunaan bahasa pemrograman Rust sebagai *backend* dan juga Laravel atau Vue.js sebagai *framework* dari aplikasi.

### 5.3 Refleksi

Ruang perbaikan dan pengembangan dalam membangun aplikasi berbasis website dapat difokuskan pada beberapa aspek. Pertama, manajemen dan perencanaan waktu adalah bagian utama dalam membangun sebuah aplikasi dengan waktu hanya tiga minggu. Kami menyadari bahwa manajemen dan perencanaan waktu yang lebih baik dan terstruktur dapat meningkatkan efisiensi penggerjaan dan juga menghindari tekanan waktu karena suatu *deadline* yang sudah dekat.

Selain itu, pemahaman terhadap spesifikasi yang diberikan juga harus diperhatikan. Memahami secara hati-hati dan menyeluruh tentang apa saja yang perlu dibangun, diminta, dan juga yang harus ada pada aplikasi. Hal ini dapat mencegah adanya risiko kesalahan pada pembuatan aplikasi dan juga aplikasi dapat berjalan sesuai dengan harapan. Selain itu, penempatan perhatian pada revisi spesifikasi juga harus selalu diperhatikan untuk menyesuaikan hal-hal yang perlu disesuaikan pada aplikasi.

Komunikasi tim yang berjalan dan baik juga harus dibangun dan bisa diperbaiki. Membuat sebuah ruang komunikasi untuk tim yang jelas, terbuka, dan inklusif di antara anggota tim untuk menghindari adanya miskomunikasi dan juga *update progress* dari hal yang telah dikerjakan oleh masing-masing. Diskusi yang diadakan secara reguler dan terjadwal terkait pembangunan dan pengembangan aplikasi juga dapat meningkatkan keterlibatan dan pemahaman untuk semua anggota tim.

Terakhir, evaluasi pada masing-masing anggota tim. Evaluasi dapat berbentuk umpan balik untuk antar anggota tim maupun dari asisten. Umpan balik ini dapat dimanfaatkan sebagai sarana pengembangan diri dari masing-masing anggota tim untuk perbaikan pada waktu selanjutnya dan juga sebagai media untuk terus berkembang. Selalu terbuka dan juga mau untuk mendengarkan saran serta komitmen untuk belajar dari setiap pengalaman juga dapat meningkatkan kinerja dan juga kompetensi yang dimiliki pada waktu selanjutnya.

## LAMPIRAN

### Miscellaneous (Tabel Poin)

Tabel 6.1: Poin yang dikerjakan

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	[Bonus] Membuat bonus video dan diunggah pada Youtube.	✓	
8	[Bonus] Membuat bonus algoritma pencarian <i>Bidirectional</i> .	✓	
9	[Bonus] Membuat bonus <i>Live Update</i> .	✓	
10	[Bonus] Aplikasi di- <i>containerize</i> dengan Docker	✓	
11	[Bonus] Aplikasi di- <i>deploy</i> dan dapat diakses melalui internet.	✓	

## **Github Repository**

Program (*source code*) dapat diakses pada

[https://github.com/Nayekah/Tubes2\\_Labpro-Hebat](https://github.com/Nayekah/Tubes2_Labpro-Hebat)

atau (website) pada pranala [seleksiasistenlabpro.xyz](http://seleksiasistenlabpro.xyz)

## **Video**

Video dapat ditonton pada [https://youtu.be/LcqgqYv5VLk?si=qT-NZZDkxtn\\_YCFN](https://youtu.be/LcqgqYv5VLk?si=qT-NZZDkxtn_YCFN)

## **DAFTAR PUSTAKA**

R. Munir and N. U. Maulidevi, "Breadth/Depth First Search (BFS/DFS)," <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/>, Jan. 2025. [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)

GeeksforGeeks (2022) *Illustrate the difference in peak memory consumption between DFS and BFS*, GeeksforGeeks. GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/illustrate-the-difference-in-peak-memory-consumption-between-dfs-and-bfs/> (Accessed: 11 May 2025).

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

E. M. Reingold and J. S. Tilford, "Tidier Drawings of Trees," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 2, pp. 223–228, Mar. 1981, doi: 10.1109/TSE.1981.234519. (Accessed : 10 May 2025)

## AKHIR KATA

Sekian, *deliverables* dari website yang kami bangun, semoga AC, dan semangat kak asisten :3 (Arigatou Gozaimasu, Bang Aland)!

...



“Peak IF experience”

“Jangan gengsi ga pake library hanya buat peningkatan performa yang gak signifikan”  
~ Orang yang ngotak ngatik visualisasi selama 2 hari 2 malem biar rapi

[Absolute Cinema](#) 

-> Nayaka, Fariz, Ilman.