

Laporan Tugas Besar 2

IF 2211 Strategi Algoritma



Disusun oleh kelompok Ahsan Geming:

1. Ahsan Malik Al Farisi 13523074
2. Kefas Kurnia Jonathan 13523113
3. Farrel Athalla Putra 13523118

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA - KOMPUTASI
INSTITUT TEKNOLOGI BANDUNG
JL. GANESA 10, BANDUNG, 40132
2025

Daftar Isi

Daftar Isi	2
BAB I	1
Deskripsi Tugas	1
BAB II	4
Landasan Teori	4
2.1. Dasar Teori	4
2.1.1. Dasar Teori BFS	4
2.1.2. Dasar Teori DFS	5
2.1.3. Dasar Teori Bidirectional Search	6
2.2. Aplikasi Web	7
BAB III	8
Analisis Pemecahan Masalah	8
3.1. Langkah Pemecahan Masalah	8
3.2. Pemetaan Masalah	8
3.3. Fitur Fungsional dan Arsitektur Aplikasi	8
3.3.1. Fitur Fungsional	8
3.3.2. Arsitektur Aplikasi	9
3.4. Ilustrasi Kasus	9
3.4.1. Ilustrasi BFS	10
3.4.2. Ilustrasi DFS	10
3.4.3. Ilustrasi Bidirectional	11
BAB IV	13
Implementasi dan Pengujian	13
4.1. Spesifikasi Teknis Program	13
4.1.1. BFS	13
4.1.2. DFS	16
4.1.3. Bidirectional	20
4.2. Tata Cara Penggunaan Program	24
4.3. Hasil Pengujian	25
4.3.1. Pengujian BFS	25
4.3.2. Pengujian DFS	29
4.3.3. Pengujian Bidirectional	33
4.4. Analisis Pengujian	36
BAB V	37
Penutup	37
5.1. Kesimpulan	37
5.2. Saran	37
5.3. Refleksi	37
Lampiran	38

BAB I

Deskripsi Tugas



Gambar 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 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

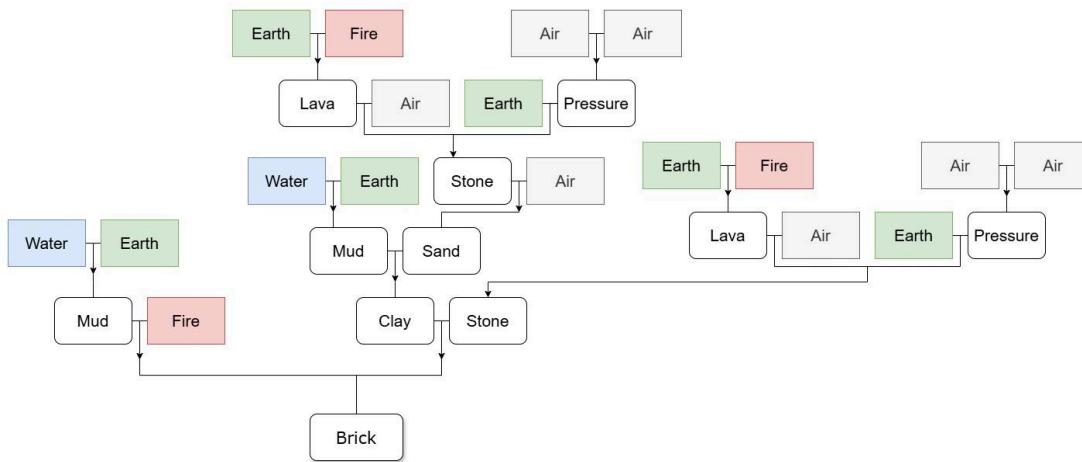
Untuk mendapatkan elemen turunan pemain dapat melakukan combine antara 2 elemen untuk menghasilkan elemen baru. Elemen turunan yang telah didapatkan dapat digunakan kembali oleh pemain untuk membentuk elemen lainnya.

Spesifikasi Wajib

- Buatlah aplikasi pencarian recipe elemen dalam permainan Little Alchemy 2 dengan menggunakan strategi BFS dan DFS.
- Tugas dikerjakan berkelompok dengan anggota minimal 2 orang dan maksimal 3 orang, boleh lintas kelas dan lintas kampus.
- Aplikasi berbasis web, untuk frontend dibangun menggunakan bahasa Javascript dengan framework Next.js atau React.js, dan untuk backend menggunakan bahasa Golang.
- Untuk repository frontend dan backend diperbolehkan digabung maupun dipisah.
- Untuk data elemen beserta resep dapat diperoleh dari scraping website Fandom Little Alchemy 2.
- Terdapat opsi pada aplikasi untuk memilih algoritma BFS atau DFS (juga bidirectional jika membuat bonus)
- Terdapat toggle button untuk memilih untuk menemukan sebuah recipe (Silahkan yang mana saja) atau mencari banyak recipe (multiple recipe) menuju suatu elemen tertentu. Apabila pengguna ingin mencari banyak recipe maka terdapat cara bagi pengguna untuk

memasukkan parameter banyak recipe maksimal yang ingin dicari. Aplikasi boleh mengeluarkan recipe apapun asalkan berbeda dan memenuhi banyak yang diinginkan pengguna (apabila mungkin).

- Mode pencarian multiple recipe wajib dioptimasi menggunakan multithreading.
- Elemen yang digunakan pada suatu recipe harus berupa elemen dengan tier lebih rendah dari elemen yang ingin dibentuk.
- Aplikasi akan memvisualisasikan recipe yang ditemukan sebagai tree yang menunjukkan kombinasi elemen yang diperlukan dari elemen dasar. Agar lebih jelas perhatikan contoh berikut.



Gambar 3. Contoh visualisasi recipe elemen

Gambar diatas menunjukkan contoh visualisasi recipe dari elemen Brick. Setiap elemen bersebelahan menunjukkan elemen yang perlu dikombinasikan. Amati bahwa leaf dari tree selalu berupa elemen dasar. Apabila dihitung, gambar diatas menunjukkan 5 buah recipe untuk Brick (karena Brick dapat dibentuk dengan kombinasi Mud+Fire atau Clay+Stone, begitu pula Stone yang dapat dibentuk oleh kombinasi Lava+Air atau Earth+Pressure). Visualisasi pada aplikasi tidak perlu persis seperti contoh diatas, tetapi pastikan bahwa recipe ditampilkan dengan jelas.

- Aplikasi juga menampilkan waktu pencarian serta banyak node yang dikunjungi.

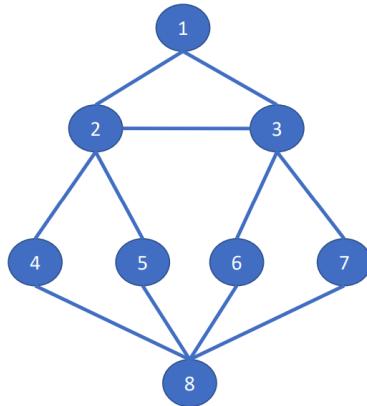
BAB II

Landasan Teori

2.1. Dasar Teori

2.1.1. Dasar Teori BFS

BFS atau Breadth First Search merupakan salah satu metode penjelajahan graf ketika tidak ada informasi tambahan yang disediakan. BFS dapat disebut juga sebagai Pencarian Melebar, hal ini karena ia akan mengunjungi semua simpul yang bertetangga dengannya, dan setiap kali suatu node dikunjungi, maka node tetangganya akan dimasukkan ke dalam struktur Queue untuk kemudian dijelajahi lebih lanjut hingga seluruh node telah dijelajahi.

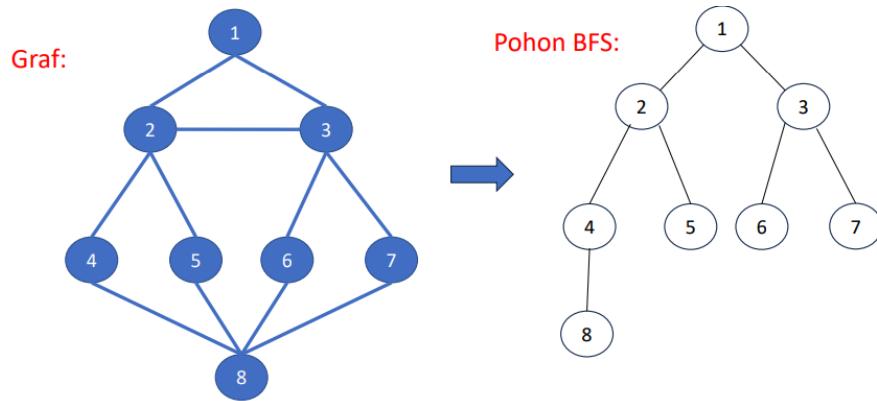


Gambar 4. Contoh Pohon Untuk Traversal BFS

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

Secara umum algoritmanya adalah seperti berikut:

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



Gambar 5. Contoh Traversasi BFS

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

Dari pohon tersebut dapat diurutkan urutan penjelajahannya ketika dimulai dari simpul 1 adalah 1, 2, 3, 4, 5, 6, 7, 8.

Secara umum algoritma BFS akan menemukan solusi terpendek dikarenakan metode ini akan menjelajahi seluruh node yang ada, namun terdapat kekurangannya yaitu bahwa metode ini memerlukan ruang memori yang banyak dan waktu traversal yang cenderung lama, sehingga pada beberapa kasus menjadi tidak efektif dalam mencari solusi terbaik. Kompleksitas waktu dari BFS adalah $O(V + E)$ dengan V adalah jumlah simpul dan E adalah jumlah edge dari suatu graph.

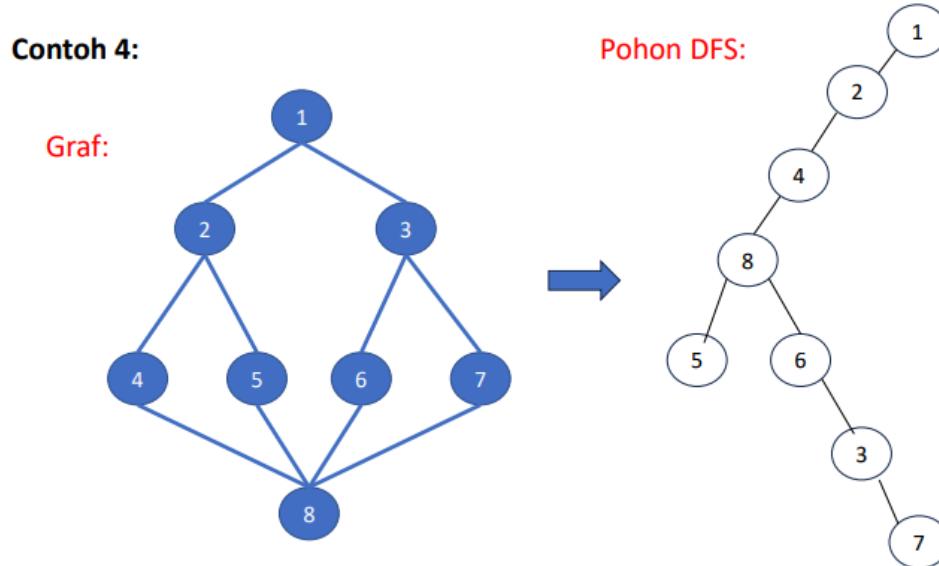
2.1.2. Dasar Teori DFS

Depth-First Search (DFS) adalah salah satu metode pencarian solusi berbasis graf tanpa informasi tambahan yang disediakan (*uninformed/blind search*). DFS akan menelusuri struktur data graf dan pohon dengan menjelajahi sebuah simpul (*node*) terlebih dahulu, kemudian secara rekursif menjelajahi simpul-simpul tetangganya, sebelum kembali (*backtrack*) ke simpul sebelumnya. Proses ini akan terus dilakukan hingga semua simpul yang dapat dicapai telah dikunjungi.

Secara garis besar, metode pencarian mendalam (DFS) memiliki algoritma sebagai berikut:

1. Awalnya, kunjungi sebuah simpul v , kemudian kunjungi simpul w yang bertetangga dengan simpul v .
2. Ulangi pencarian DFS dari simpul w .
3. Ketika mencapai simpul u sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, *backtrack* ke simpul terakhir yang dikunjungi sebelumnya dan memiliki simpul w yang belum dikunjungi
4. Pencarian dinyatakan berakhir jika tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.

Sebagai visualisasi, proses algoritma pencarian mendalam ini dapat dalam graf dan pohon DFS sebagai berikut:



Gambar 6. Contoh Traversal Graf Secara DFS

Sumber: [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)

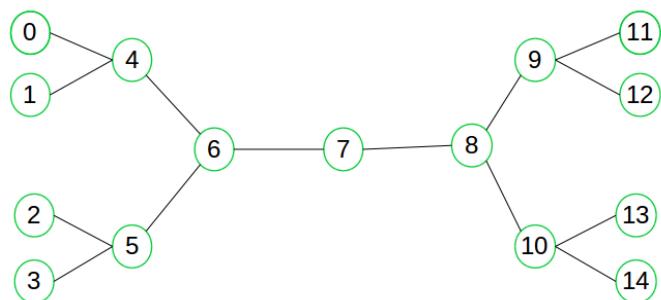
Pada contoh ini, urutan simpul-simpul yang dikunjungi secara DFS dari 1 adalah: 1, 2, 4, 8, 5, 6, 3, 7.

DFS disebut “*depth-first*” karena penelusurannya dilakukan sedalam mungkin sebelum kembali ke level sebelumnya. Sehingga, metode pencarian DFS ini cocok untuk mengeksplorasi semua kemungkinan atau jalur (seperti dalam pencarian solusi). Algoritma ini memiliki kompleksitas waktu $O(V+E)$ untuk graf berukuran V simpul dan E sisi. Dalam kehidupan sehari-hari, algoritma ini digunakan dalam *citation map*, *web spider*, beberapa pencarian untuk permainan, termasuk permainan Little Alchemy 2.

2.1.3. Dasar Teori Bidirectional Search

Bidirectional search merupakan salah satu metode penjelajahan graf yang memiliki perbedaan unik dibandingkan BFS dan DFS. BFS dan DFS memiliki arah penjelajahan dari simpul asal ke simpul tujuan, namun Bidirectional memulai traversal dari kedua arah secara sekaligus. Sehingga algoritma ini mencari path tercepat dari simpul asal ke simpul tujuan.

Algoritma dari Bidirectional ini menjalankan dua pencarian secara sekaligus, yaitu *forward search* dari simpul asal ke simpul tujuan dan *backward search* dari simpul tujuan ke simpul asal. Pencarian berhenti ketika kedua graf beririsan.



Gambar 7. Contoh Pohon Bidirectional

Ketika terdapat path dari simpul 0 ke 14. Maka pencarian dilakukan dari dua sisi dan keduanya akan bertemu di simpul 7. Dengan ini kita dapat mengeliminasi banyak pencarian yang tidak perlu.

Penggunaan bidirectional banyak digunakan karena algoritmanya yang lebih cepat dan secara signifikan mengurangi jumlah pencarian yang dilakukan. Algoritma ini dapat dipakai ketika tujuan awal dan akhirnya unik dan terdefinisi, juga ketika faktor percabangan sama di kedua arah.

Dari segi performa, bidirectional dapat menjadi complete jika memakai BFS di kedua sisi. Bidirectional juga dapat menjadi optimal jika BFS dipakai dan seluruh pathnya memiliki cost yang sama. Kompleksitas waktu dari bidirectional adalah $O(b^{\frac{d}{2}})$ dengan b adalah branching factor dan d adalah jarak dari simpul tujuan dari simpul asal.

2.2. Aplikasi Web

Aplikasi web ini dirancang untuk membantu pengguna menemukan resep elemen dengan berbagai mode pencarian, termasuk BFS (Breadth-First Search), DFS (Depth-First Search), dan Bidirectional Search. Pengguna dapat menyesuaikan jumlah resep yang ditampilkan, melihat visualisasi graf secara realtime, dan menelusuri detail setiap langkah pencarian melalui sub-halaman yang dihasilkan. Selain itu, aplikasi ini menghasilkan output dalam bentuk JSON dan menampilkan daftar pembentukan elemen, seperti Stone + Wind = Sand.

Dibuat menggunakan Next.js untuk frontend dan Go untuk backend, aplikasi ini dikemas dalam Docker untuk memastikan portabilitas, konsistensi, dan skalabilitas, menjadikannya solusi yang efisien untuk pencarian resep elemen secara interaktif.

BAB III

Analisis Pemecahan Masalah

3.1. Langkah Pemecahan Masalah

Pertama-tama, untuk menyelesaikan masalah ini, kita perlu memahami alur pembentukan elemen dari berbagai resep dan bagaimana mencapai target elemen dari elemen-elemen awal. Langkah-langkah utamanya meliputi:

1. Identifikasi target dan element awal, dengan cara menentukan target element sebagai titik awal dan elemen-elemen awal (leaf) sebagai titik akhir dari proses yang ingin dicapai.
2. Membangun struktur data untuk representasi masalah yaitu dengan memetakan setiap elemen dan hubungannya dalam bentuk graf dengan elemen sebagai node dan relasi recipe sebagai edge. Selain dari itu juga memastikan struktur data yang dipakai dapat menunjang penjelajahan graf dengan efisien.
3. Pencarian jalur resep dengan BFS, DFS ataupun Bidirectional semuanya disusun algoritmanya agar dapat mencari recipe baik tunggal maupun multiple dengan constraint bahwa elemen harus dibentuk oleh elemen lain dengan tier yang lebih kecil.
4. Pembatasan lintasan untuk mencegah loop dengan menandai elemen mana yang sudah dikunjungi.
5. Pembatasan eksplorasi recipe agar tidak terjadi ledakan dengan memberikan constraint.
6. Mengonversi hasil scraping elemen menjadi map yang berisi recipe untuk membuat tiap elemen.

3.2. Pemetaan Masalah

1. Element: Node
2. Elemen target: Root Node
3. Elemen awal: Leaf
4. Relasi antara elemen dengan element di dalam resep: Edge
5. Jalur Resep: Path
6. Pembatasan lintasan: Path Pruning

3.3. Fitur Fungsional dan Arsitektur Aplikasi

3.3.1. Fitur Fungsional

- a. Pencarian resep

Terdapat beberapa fitur utama yang dirancang untuk memudahkan pengguna dalam menelusuri hubungan antar elemen. Pengguna dapat memilih mode pencarian, termasuk BFS (Breadth-First Search), DFS (Depth-First Search), atau Bidirectional Search, sesuai dengan kebutuhan pencarian. Selain itu, pengguna juga dapat menentukan jumlah resep yang ingin ditampilkan untuk mempersempit hasil pencarian.

b. Visualisasi hasil

Hasil akan divisualisasikan dalam bentuk graf secara realtime, menampilkan elemen-elemen yang terhubung secara dinamis. Setiap jalur pencarian akan ditampilkan dalam bentuk sub-halaman yang bisa diakses pengguna untuk melihat detail dari setiap langkah pencarian. Selain itu agar lebih mudah memahami akan bagaimana elemen dibentuk, aplikasi juga menyediakan daftar pembentukan elemen seperti Stone + Wind = Sand atau Earth + Pressure = Stone yang menampilkan hubungan antar elemen secara langsung. Selain visualisasi, aplikasi ini juga memperlihatkan output dalam bentuk JSON.

c. Hasil dari pencarian juga akan menampilkan beberapa informasi tambahan seperti waktu eksekusi dan jumlah node yang dikunjungi.

3.3.2. Arsitektur Aplikasi

Aplikasi ini dibangun dengan arsitektur yang memisahkan frontend dan backend untuk meningkatkan skalabilitas dan efisiensi. Next.js digunakan sebagai frontend dengan fitur server-side rendering untuk memastikan performa optimal. Backend aplikasi dikembangkan menggunakan Go (Golang) untuk memproses logika pencarian secara efisien, termasuk penerapan algoritma BFS, DFS, dan Bidirectional Search yang dirancang untuk menangani jumlah node yang besar dengan cepat.

Untuk memastikan portabilitas dan kemudahan pengelolaan aplikasi, seluruh layanan dikemas dalam Docker container, yang memungkinkan aplikasi untuk dijalankan secara konsisten di berbagai lingkungan.

3.4. Ilustrasi Kasus

Misalkan kita di dalam aplikasinya dan kita ingin untuk mencari elemen Clay sebanyak 2. Maka Clay akan memiliki beberapa resep yang di dalamnya akan memiliki beberapa resep lagi yang dapat dicari, seperti:

- Resep 1: Mud + Sand.
- Resep 2: Mud + Stone.
- Resep 3: Mineral + Sand

- Resep 4: Mineral + Stone
- Resep 5: Mineral + Rock
- Resep 6: Stone + Liquid
- Resep 7: Rock + Liquid

Maka dari sini setiap bahan juga akan memiliki resepnya sendiri, seperti:

1. Mud: Water + Earth, Water + Soil.
2. Sand: Stone + Air, Stone + Wind, Air + Rock, Wind + Rock, Pebble + Air, Pebble + Wind, Pebble + Small.
3. Stone: Earth + Pressure, Earth + Solid, Air + Lava
4. Mineral: Organic Matter + (Stone/Rock/Earth Boulder/Hill/Mountain)
5. Liquid: Water + Science, Water + Idea, Energy + Solid.

3.4.1. Ilustrasi BFS

Metode BFS akan dimulai dari node Clay sebagai akar pencarian. Pada langkah pertama, algoritma akan mencari semua resep yang terkait dengan Clay, yang berjumlah 7 resep. Setiap resep ini kemudian diseleksi menggunakan heuristik, yaitu dengan membandingkan tier setiap elemen dalam resep tersebut. Jika elemen dalam resep memiliki tier yang lebih tinggi dari Clay, maka resep tersebut akan diabaikan. Sebaliknya, jika tier elemen tersebut sama atau lebih rendah dari Clay, resep tersebut akan dianggap valid untuk diproses lebih lanjut.

Sebelum eksplorasi dilanjutkan, resep yang telah lolos seleksi akan diurutkan berdasarkan total tier-nya, dengan resep yang memiliki total tier terendah diprioritaskan untuk dieksplorasi terlebih dahulu. Karena kita ingin mencari setidaknya dua resep yang berbeda, resep valid kedua akan dimasukkan ke dalam antrean BFS sebagai cabang alternatif. Sebelum memasukkan cabang ini, status BFS saat ini akan disalin untuk memastikan eksplorasi berjalan independen. Setelah jalur utama selesai diproses, cabang alternatif ini akan dilanjutkan.

Hasil akhir dari proses ini adalah dua jalur yang ditemukan, yaitu Mud + Stone (Earth + Pressure) dan Mud + Stone (Air + Lava).

3.4.2. Ilustrasi DFS

Menggunakan DFS, akan dimulai dari node Clay juga sebagai akar pencarian. Program kemudian akan meluncurkan beberapa *goroutine* untuk base recipe (Mud + Sand, Mud + Stone, dll) karena menggunakan *multithreading*. Setiap *goroutine* akan melakukan eksplorasi secara DFS dari *initial path* masing-masing. Untuk setiap resep, program akan

membuat tree untuk dieksplorasi. Sama seperti BFS, setiap resep ini akan diseleksi menggunakan heuristik, yaitu membandingkan setiap tier dari elemen-elemen yang akan diujikan. Jika suatu resep memiliki tier lebih tinggi dari elemen yang akan dibuatnya, maka tidak akan dipilih untuk menghindari *cyclic dependencies*. Dalam kasus ini, resep yang lebih rendah tiernya daripada Clay akan dianggap valid.

Selanjutnya, untuk setiap elemen yang bukan elemen dasar (seperti Mud), program akan mengeksplorasi beberapa cara berbeda untuk membuatnya. Seperti DFS, untuk setiap node akan ditelusuri hingga paling bawah hingga elemen dasar. Kemudian, *backtrack* untuk menemukan kemungkinan solusi lainnya yang berbeda node tetapi membangun resep yang sama dengan tetap memperhatikan aturan tier sebelumnya. Elemen yang sudah pernah dikunjungi sebelumnya akan disimpan untuk menghindari terjadinya siklus dan optimasi program.

Pencarian akan dihentikan ketika program sudah menemukan jumlah resep yang diinginkan pengguna, atau sudah tidak ada lagi kemungkinan resep lainnya. Pada kasus Clay, dari ketujuh resep utama, yang akan dicari hanya melalui Mud + Stone karena resep lainnya berada pada tier yang lebih tinggi. Mud hanya bisa dicari menggunakan Earth + Water (*basic elements*), sedangkan Stone dapat dibuat menggunakan Earth + Pressure dan Air + Lava. Pressure hanya dapat dibuat menggunakan Air + Air (*basic elements*) dan Lava dari Earth + Fire (*basic elements*). Sehingga, hasil akhirnya adalah dua jalur berbeda ditemukan untuk Clay, yaitu Mud + Stone, dengan yang satu Stone menggunakan Air + Lava, yang satu lagi menggunakan Earth + Pressure.

3.4.3. Ilustrasi Bidirectional

Jika algoritma bidirectional search (BDS) digunakan untuk mencari resep elemen seperti *Clay*, maka proses pencarian akan dimulai dari dua arah sekaligus: arah *forward* dari elemen-elemen dasar seperti *Water*, *Earth*, atau *Fire*, dan arah *backward* dari target yaitu *Clay* itu sendiri. Dalam pencarian backward, sistem akan mengakses semua resep yang dapat membentuk *Clay*, misalnya *Mud + Sand*, *Mud + Stone*, dan sebagainya. Untuk setiap bahan dalam resep tersebut, algoritma akan memeriksa apakah bahan tersebut telah ditemukan oleh pencarian forward. Jika ditemukan bahwa semua bahan dalam sebuah resep telah tersedia dalam forward search, maka dianggap terjadi titik temu (*meeting point*), dan jalur tersebut dianggap valid untuk membentuk *Clay*. Pada saat yang sama, pencarian forward akan secara bertahap membentuk berbagai elemen dari kombinasi bahan dasar. Semua jalur disaring dengan heuristik berdasarkan *tier*; artinya, jika sebuah resep menggunakan bahan yang tier-nya lebih tinggi dari target, maka resep itu akan diabaikan demi menghindari *cyclic dependencies*.

Setelah titik temu ditemukan, algoritma akan mulai membangun pohon resep (recipe tree) secara dari target elemen menuju bahan-bahan pembentuknya, baik dari arah backward

maupun forward. Proses ini memanfaatkan struktur data seperti RecipeNode dan BidirRecipePath untuk merekam setiap jalur pembuatan dan kedalaman (depth) dari elemen tersebut dalam hirarki penciptaan. Untuk mencegah terjadinya *infinite loop*, algoritma menyimpan status kunjungan dalam peta visited dan membatasi kedalaman eksplorasi hingga batas tertentu (misalnya 15–20 langkah). Jika ditemukan elemen yang sudah pernah dikunjungi atau mencapai kedalaman maksimum, algoritma akan menghentikan eksplorasi cabang tersebut.

BAB IV

Implementasi dan Pengujian

4.1. Spesifikasi Teknis Program

4.1.1. BFS

a. Struktur Data

Struktur Data	Keterangan
<pre>type Element struct { Element string Recipes []string }</pre>	Menyimpan data satu elemen dari parsing json.
<pre>type ElementData map[string][]Element</pre>	Menyimpan kumpulan elemen dari parsing json.
<pre>type ProcessUpdate struct { Type string Element string Path interface{} Complete bool Stats struct { NodeCount int StepCount int } }</pre>	Menyimpan state path untuk dikirim melalui web socket.
<pre>type WebSocketClient struct { conn *websocket.Conn mu sync.Mutex }</pre>	WebSocketClient wrap koneksi WebSocket dengan method thread-safe.
<pre>type RecipeNode struct { Element string Recipes []*RecipeNode }</pre>	Menyimpan node dari elemen saat dilakukan penjelajahan BFS dengan struktur Tree.
<pre>type RecipeInfo struct { Recipe []string TotalTier int }</pre>	Menyimpan informasi terdiri dari elemen apa dan jumlah tier nya

}	berapa.
<pre>type ElementToProcess struct { Element string Depth int }</pre>	Struktur yang disimpan di dalam queue untuk kemudian di proses oleh BFS.
<pre>type RecipePath struct { Elements []string Recipes map[string][]string }</pre>	Tracking elemen apa saja yang sudah dicari dan apa resepnya dalam bentuk map.
<pre>type WorkItem struct { Path RecipePath Queue []ElementToProcess }</pre>	Struktur untuk dibuat channel saat multithreading yang menyimpan hasil dari seluruh node yang ada.

b. Fungsi

Fungsi	Keterangan
<pre>func buildRecipeMap(data ElementData) map[string][][]string</pre>	Mengambil kumpulan elemen dari hasil parsing untuk kemudian dibuat struktur data map dari kumpulan resep suatu elemen untuk memudahkan pencarian.
<pre>func buildTierMap(data ElementData) map[string]int</pre>	Mengambil kumpulan elemen dari hasil parsing untuk kemudian dibuat struktur data map dari kumpulan elemen ke tier nya.
<pre>func ConvertToJson(nodes []*RecipeNode) string</pre>	Mengubah array of Recipe Node menjadi output string.
<pre>func findRootElement(path RecipePath) string</pre>	Mencari root element dari suatu path.

<pre>func buildRecipeTreeFromPath (path RecipePath, element string, startingElements map[string]bool) *RecipeNode</pre>	Membuat Recipe Node dari suatu path Recipe Path.
<pre>func getValidRecipes(recipeMap map[string][][]string, tierMap map[string]int, element string) []RecipeInfo</pre>	Mencari resep dari suatu elemen dan menyeleksinya dengan konstrain tier yang ada.
<pre>func copyRecipePath(path RecipePath) RecipePath</pre>	Menyalin suatu Recipe Path.
<pre>func containsString(slice []string, str string) bool</pre>	Mengecek apakah terdapat string yang sesuai pada slice atau array.
<pre>func BFSSingleWithUpdates(target Element string, recipeMap map[string][][]string, tierMap map[string]int, conn *websocket.Conn) interface{}</pre>	Mencari resep dengan BFS jika hanya satu elemen yang dicari sambil melakukan koneksi ke websocket untuk mengirim update path.
<pre>func BFSMultipleWithUpdates(targetElement string, recipeMap map[string][][]string, tierMap map[string]int, recipeLimit int, conn *websocket.Conn) interface{}</pre>	Sebagai pengelola utama untuk pencarian jalur resep menggunakan metode BFS. Fungsi ini akan mengambil initial root path dari elemen dan juga memulai goroutine dari channel. Kemudian akan dimulai pekerjaan secara paralel yang akan memanggil processPath yang hasilnya akan dikumpulkan melalui resultChan <- *RecipeNode
<pre>func (c *WebSocketClient) SendUpdate(update</pre>	Mengirimkan update state ke web

ProcessUpdate) error	melalui web socket.
----------------------	---------------------

c. Prosedur

Prosedur	Keterangan
<pre>func processPathMultiple(current Path RecipePath, queue []ElementToProcess, recipeMap map[string][][]string, tierMap map[string]int, workChan chan<- WorkItem, resultChan chan<- *RecipeNode, branchCount *int, recipeLimit int)</pre>	Merupakan worker untuk memproses setiap jalur secara terpisah. Fungsi ini memproses setiap elemen dan mencari resepnya, di setiap saat pengecekan elemen terjadi branching jika terdapat beberapa resep yang dapat dipilih. Jika terjadi branching maka path yang sekarang akan di-copy dan dimasukkan ke loop BFS untuk kemudian nanti dilakukan traversal ulang. Hasil dari ini akan disimpan melalui resultChan.
<pre>func processVisualizationPath(cu rrentPath RecipePath, queue []ElementToProcess, recipeMap map[string][][]string, tierMap map[string]int, client *WebSocketClient, nodeCount *int32, stepCount *int32)</pre>	Prosedur untuk mengirimkan update visualisasi path ke web socket.

4.1.2. DFS

a. Struktur Data

Struktur Data	Keterangan
<pre>type Element struct { Element string `json:"element"` Recipes []string</pre>	Menyimpan data elemen dari parsing JSON yang ada.

<pre>`json:"recipes"` }</pre>	
<pre>type ElementData map[string][]Element</pre>	Menyimpan kumpulan elemen dari parsing json.
<pre>type RecipeNode struct { Element string `json:"element"` Recipes []*RecipeNode `json:"recipes"` }</pre>	Menyimpan node dari elemen saat dilakukan penjelajahan DFS dengan struktur Tree.
<pre>type ingredient struct { path []int element string }</pre>	Struktur untuk melacak posisi bahan dalam <i>recipe tree</i> .
<pre>type explorationKey struct { element string recipe string path string }</pre>	Kunci untuk menghindari pengulangan eksplorasi
<pre>type workItem struct { node *RecipeNode path []int depth int }</pre>	Struktur untuk dibuat channel saat multithreading yang menyimpan hasil dari seluruh node yang ada.

b. Fungsi

Fungsi	Keterangan
<pre>func buildRecipeMap(data ElementData) map[string][][]string</pre>	Mengambil kumpulan elemen dari hasil parsing untuk kemudian dibuat struktur data map dari kumpulan resep suatu elemen untuk memudahkan pencarian.

<pre>func calculateElementTiers(recipeMap map[string][][]string) map[string]int</pre>	Fungsi untuk menghitung tier dari setiap elemen dalam Little Alchemy 2 yang digunakan untuk mencegah <i>cyclic</i> dan ketentuan spesifikasi.
<pre>func findBestRecipe(element string, recipeMap map[string][][]string, maxAllowedTier int) []string</pre>	Fungsi untuk menemukan resep terbaik untuk membuat suatu elemen, memilih yang paling efisien dan sederhana untuk membuat elemen.
<pre>func filterAndSortRecipes(recipes [][]string, targetTier int) [][]string</pre>	Memfilter dan mengurutkan resep berdasarkan validitas tier, memastikan algoritma DFS menelusuri jalur resep yang valid.
<pre>func generateRecipeHash(node *RecipeNode) string</pre>	Menghasilkan representasi unik (hash) dari sebuah pohon resep supaya algoritma hanya menyimpan resep yang benar-benar unik.
<pre>func countIngredients(node *RecipeNode) int</pre>	Menghitung total jumlah bahan dalam sebuah resep, termasuk sub-resep, untuk mengukur kompleksitas resep.
<pre>func isElementAllowedForTier(element string, targetTier int) bool</pre>	Fungsi untuk memeriksa apakah suatu elemen diperbolehkan untuk digunakan dalam resep pada tier tertentu.
<pre>func buildRecipeTreeDFSWithUpdates(node *RecipeNode, recipeMap map[string][][]string, visited map[string]bool, client *WebSocketClient, targetElement string) bool</pre>	Membangun pohon <i>recipe</i> menggunakan algoritma DFS yang juga mengirimkan progres secara real-time.
<pre>func</pre>	Mencari satu resep tunggal untuk

<pre>DFSSingleWithUpdates(target Element string, recipeMap map[string][][]string, tierMap map[string]int, conn *websocket.Conn) interface{}</pre>	elemen target dan mengembalikannya
<pre>func buildInitialRecipeTree(element string, ingredients []string, recipeMap map[string][][]string, targetTier int) *RecipeNode</pre>	Fungsi untuk membangun pohon <i>recipe</i> awal untuk elemen target berdasarkan daftar bahan yang sudah ditentukan.
<pre>func buildCompleteRecipeTree(element string, recipeMap map[string][][]string, visited map[string]bool) *RecipeNode</pre>	Fungsi untuk membangun sebuah pohon <i>recipe</i> lengkap untuk suatu elemen secara rekursif.
<pre>func buildSubRecipeTree(recipe []string, element string, recipeMap map[string][][]string) *RecipeNode</pre>	Membangun <i>sub-tree</i> untuk bahan spesifik menggunakan resep yang sudah ditentukan untuk substitusi sub-bagian pohon resep dengan implementasi alternatif.
<pre>func DFSMultipleWithUpdates(targetElement string, recipeMap map[string][][]string, tierMap map[string]int, recipeLimit int, conn *websocket.Conn) interface{}</pre>	Fungsi untuk mencari beberapa variasi resep berbeda untuk membuat elemen target.

c. Prosedur

Prosedur	Keterangan
func	Membuat variasi resep dengan

<pre>createMultiPointVariation(baseRecipe *RecipeNode, recipeMap map[string][][]string, results *[]*RecipeNode, uniqueRecipes map[string]bool, mutex *sync.Mutex, recipeLimit int)</pre>	mengganti beberapa titik (<i>multi-point</i>) dalam resep yang sudah ada.
<pre>func exploreAllRecipeVariationsWithUpdates(element string, ingredients []string, recipeMap map[string][][]string, targetTier int, results *[]*RecipeNode, uniqueRecipes map[string]bool, mutex *sync.Mutex, recipeLimit int, client *WebSocketClient)</pre>	Mengeksplorasi semua variasi resep yang mungkin dari resep dasar, mengganti elemen tingkat rendah untuk menghasilkan resep berbeda.
<pre>func sortRecipesByComplexity(recipes []*RecipeNode)</pre>	Prosedur untuk mengurutkan daftar resep berdasarkan kompleksitasnya.

4.1.3. Bidirectional

a. Struktur Data

Struktur Data	Keterangan
<pre>type BidirRecipePath struct { Element string Recipe []string Depth int IsMeeting bool }</pre>	Menyimpan jalur pencarian (baik dari arah <i>forward</i> maupun <i>backward</i>) dalam pencarian bidirectional. Dengan Element adalah nama <i>element</i> yang dicapai, Recipe adalah daftar bahan pembuat element, Depth adalah level kedalaman pencarian, dan IsMeeting

	untuk mencatat apakah node ini merupakan titik pertemuan dari dua arah.
<pre>type BidirSearchData struct { ForwardVisited, BackwardVisited map[string]BidirRecipePath ForwardQueue, BackwardQueue []string MeetingPoints []string MeetingElements map[string]bool NodeCount, StepCount int }</pre>	<p>Menyimpan semua status selama pencarian bidirectional.</p> <p>ForwardVisited dan BackwardVisited menyimpan node yang telah dikunjungi. ForwardQueue dan BackwardQueue menyimpan node yang akan dikunjungi. MeetingPoints dan MeetingElements mencatat titik temu antara dua arah pencarian.</p> <p>NodeCount dan StepCount untuk mencatat statistik dari pencarian.</p>
<pre>type WebSocketResponse struct { Element string RecipeCount int Recipes []*RecipeNode ForwardVisited, BackwardVisited map[string]BidirRecipePath MeetingPoints []string }</pre>	<p>Menyusun data lengkap yang akan dikirimkan ke frontend melalui WebSocket. Data terdiri dari nama element, tree recipenya, jejak pencarian dua arah, dan titik temu.</p>
<pre>type IngredientInfo struct { node *RecipeNode path []int }</pre>	Mencari posisi bahan dalam pohon recipe untuk dilakukan variasi.

b. Fungsi

Fungsi	Keterangan
<pre>func filterBidirValidRecipes(recipes [][]string, tierMap map[string]int, targetTier int) [][]string</pre>	Menyaring resep yang tidak mengandung element yang tidak diperbolehkan (seperti Time dan Babe the Blue Ox) dan hanya mengambil bahan yang berada di tier lebih rendah.
<pre>func calculateBidirTierSum(recipe []string, tierMap map[string]int) (int, int)</pre>	Menghitung jumlah tier dari semua bahan dalam suatu resep dan jumlah bahan yang digunakan selama proses.
<pre>func SafeBidirectionalSingleWithUpdates(targetElement string, recipeMap map[string][][]string, tierMap map[string]int, conn *websocket.Conn) interface{}</pre>	Mencari satu recipe pembuatan suatu elemen menggunakan pencarian bidirectional serta mengirimkan hasil ke frontend secara bertahap melalui WebSocket.
<pre>func performBidirectionalSearchWithTimeout(data *BidirSearchData, targetElement string, recipeMap map[string][][]string, tierMap map[string]int, client *WebSocketClient) bool</pre>	Menjalankan algoritma pencarian bidirectional dengan batas waktu, depth, dan node.
<pre>func buildBidirRecipeTreeSafe(element string, data *BidirSearchData, recipeMap map[string][][]string, tierMap map[string]int, depth int, maxDepth int, visited map[string]bool)</pre>	Membangun <i>tree recipe</i> dari data pencarian dan menghindari <i>cycle</i> dan <i>infinite loop</i> .

*RecipeNode	
func SafeBidirectionalMultipleWi thUpdates(targetElement string, recipeMap map[string][][]string, tierMap map[string]int, recipeLimit int, conn *websocket.Conn) interface{ }	Mencari banyak variasi jalur pembuatan suatu elemen dengan pendekatan bidirectional dan mengirimkan hasil ke frontend secara bertahap melalui WebSocket.
func filterDuplicateRecipes(reci pes []*RecipeNode) []*RecipeNode	Menghilangkan recipe yang duplikat sehingga hanya recipe unik yang tersisa.
func deepCopyRecipeNode(node *RecipeNode) *RecipeNode	Membuat <i>deep copy</i> dari sebuah node agar tidak mengubah data asli.
func completeRecipeTree(element string, recipe []string, recipeMap map[string][][]string, tierMap map[string]int, visited map[string]bool) *RecipeNode	Membangun pohon resep penuh dari satu resep root tertentu.
func findNonBasicIngredients(rec ipe *RecipeNode) []IngredientInfo	Mencatat semua bahan yang bukan <i>starting element</i> ke dalam suatu pohon.
func getRecipeSignature(recipe *RecipeNode) string	Mengambil <i>signature</i> unik untuk membedakan tree resep.
func createCompleteSearchData(re cipes []*RecipeNode,	Menggabungkan hasil forward dan backward ke dalam struktur search data lengkap.

targetElement string) *BidirSearchData	
func findAllUniqueVariations (bas eRecipes []*RecipeNode, recipeMap map[string][][]string, tierMap map[string]int, maxCount int) []*RecipeNode	Melakukan eksplorasi variasi resep pohon secara mendalam.
func findDeepUniqueVariations (ba seRecipes []*RecipeNode, recipeMap map[string][][]string, tierMap map[string]int, count int) []*RecipeNode	Melakukan eksplorasi lanjutan untuk menemukan variasi resep yang lebih dalam.
func compareRecipeTrees (tree1, tree2 *RecipeNode) bool	Membandingkan dua pohon resep (RecipeNode) dan menentukan apakah struktur mereka sama pada level saat ini.

c. Prosedur

Prosedur	Keterangan
func buildSignature (node *RecipeNode, sb *strings.Builder, depth int)	Membuat representasi string unik dari pohon resep.
func processCompleteTree (node *RecipeNode, data *BidirSearchData, depth int)	Mengisi data BackwardVisited dari pohon resep.
func applyRecipeChange (root *RecipeNode, path []int,	Mengganti satu node bahan dalam sebuah pohon resep dengan resep

```

element string, recipe
[]string, recipeMap
map[string][][]string,
tierMap map[string]int)

```

alternatif baru.

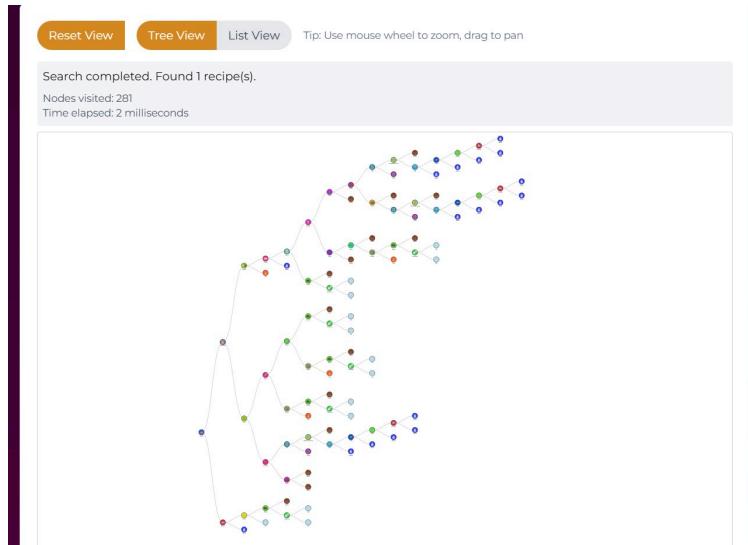
4.2. Tata Cara Penggunaan Program

- Akses web aplikasi pada <https://tubes2-fe-ahsan-geming.vercel.app/>.
- Masukkan elemen yang ingin dicari pada kolom search yang tersedia.
- Pilih metode penjelajahan graf yang diinginkan (BFS/DFS/Bidirectional).
- Pilih jumlah resep yang ingin didapatkan.
- Pencet tombol pencarian dan tunggu hasilnya.
- Lihat hasil resep realtime yang ditemukan dan juga list akan elemen apa saja yang saling membentuk satu sama lain, selain itu dapat dilihat juga hasil output json-nya, waktu eksekusi, dan jumlah node yang dikunjungi. Jika terdapat beberapa resep maka pengguna dapat navigasi page resep yang telah didapat.

4.3. Hasil Pengujian

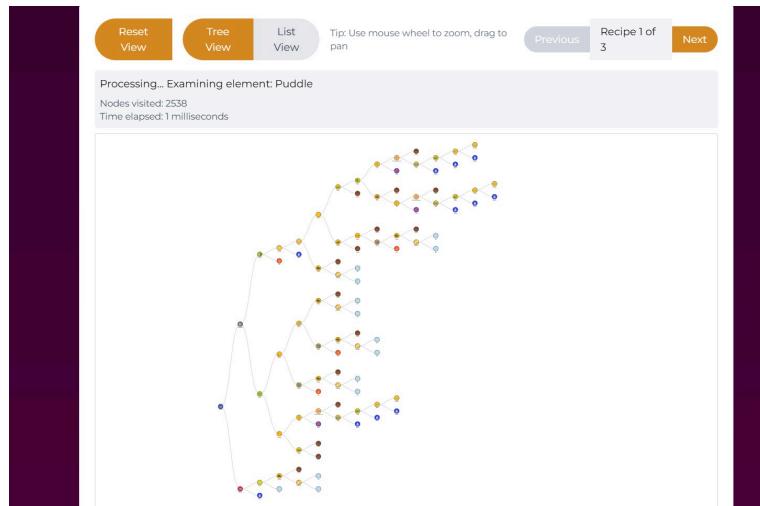
4.3.1. Pengujian BFS

- Elemen Picnic
 - Single

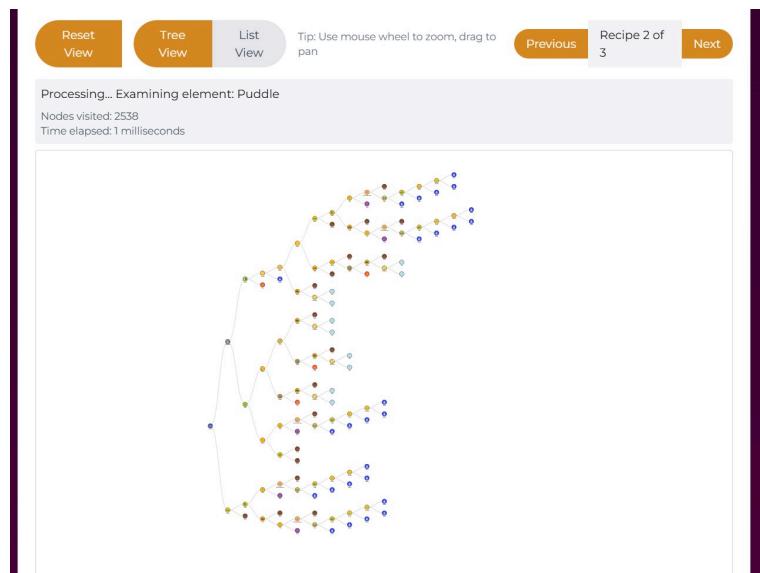


Gambar 8. Pengujian BFS Picnic Single

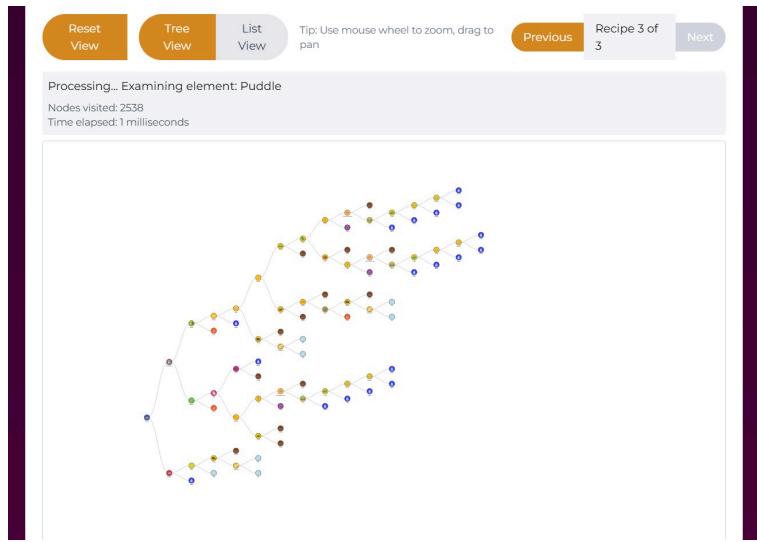
- Multiple



Gambar 9. Pengujian BFS Picnic Multiple (1/3)

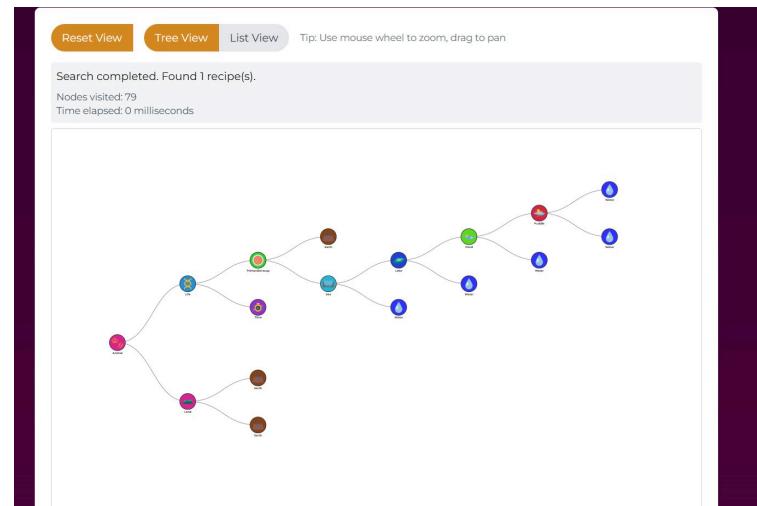


Gambar x. Pengujian BFS Picnic Multiple (2/3)



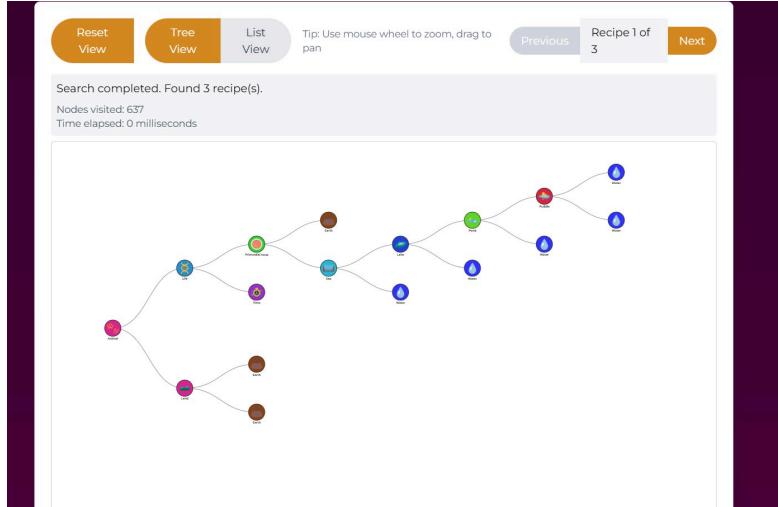
Gambar 10. Pengujian BFS Picnic Multiple (3/3)

b. Elemen Animal
i. Single

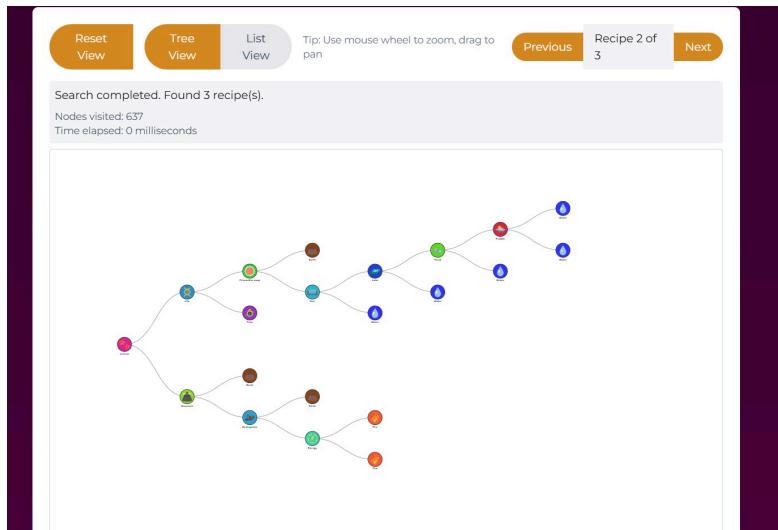


Gambar 11. Pengujian BFS Animal Single

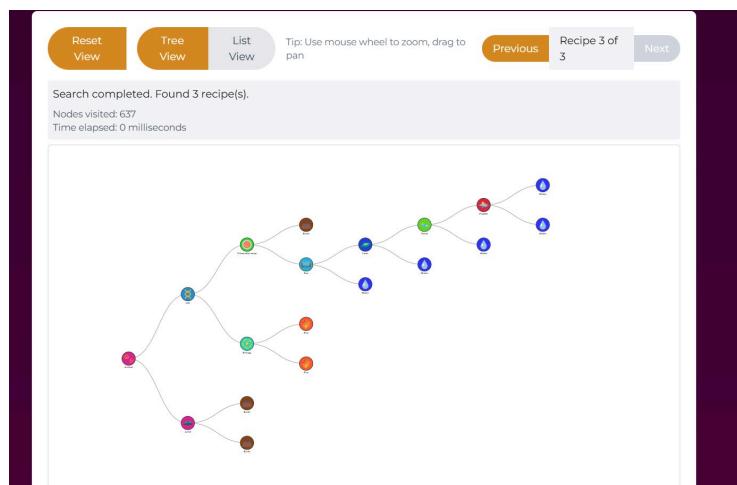
ii. Multiple



Gambar 12. Pengujian BFS Animal Multiple (1/3)

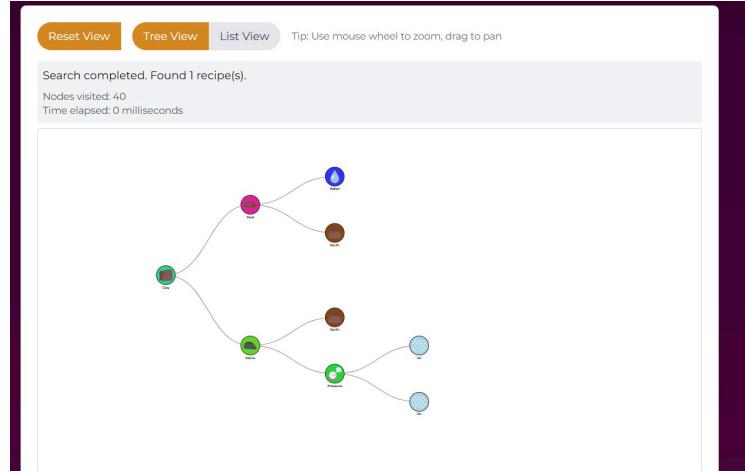


Gambar 13. Pengujian BFS Animal Multiple (2/3)



Gambar 14. Pengujian BFS Animal Multiple (3/3)

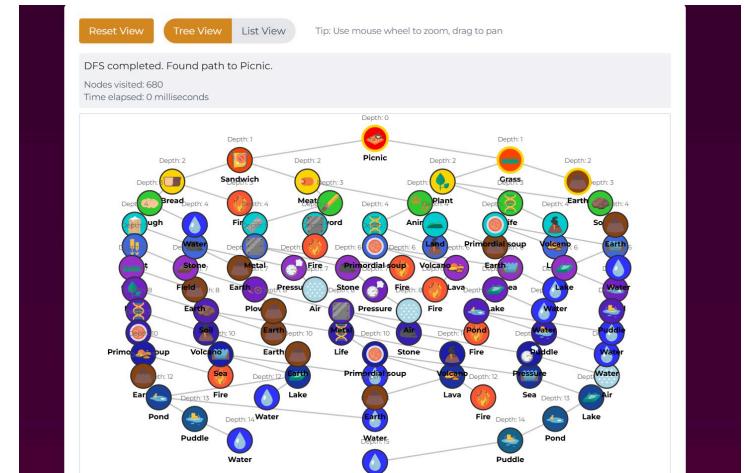
- c. Elemen Clay
 - i. Single



Gambar 15. Pengujian BFS Clay Single

4.3.2. Pengujian DFS

- a. Elemen Picnic
 - i. Single

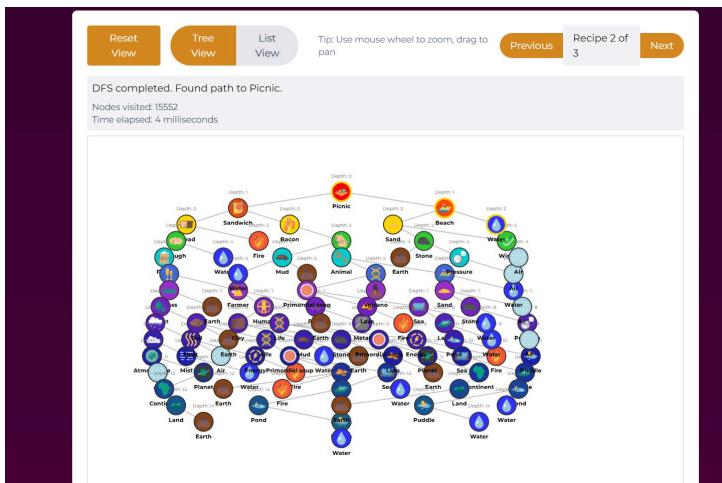


Gambar 16. Pengujian DFS Picnic Single

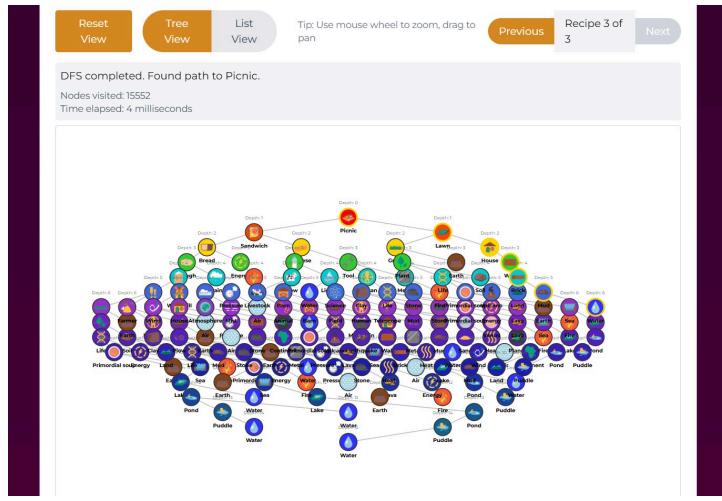
ii. Multiple



Gambar 17. Pengujian DFS Picnic Multiple (1/3)



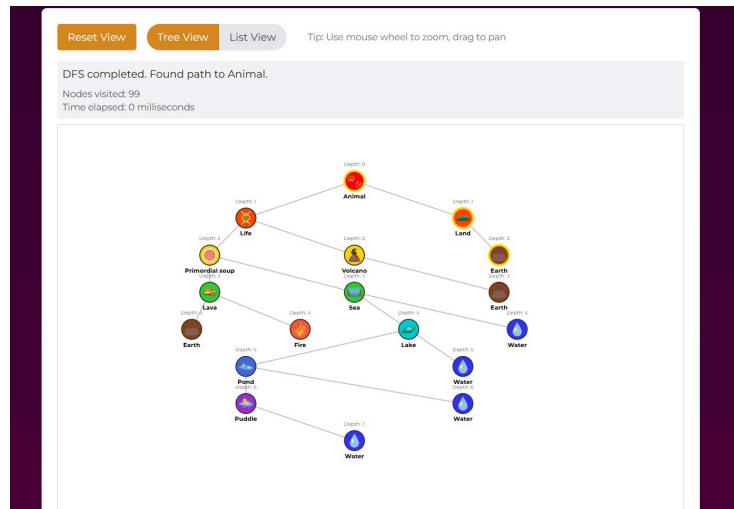
Gambar 18. Pengujian DFS Picnic Multiple (2/3)



Gambar 19. Pengujian DFS Picnic Multiple (3/3)

b. Elemen Animal

i. Single

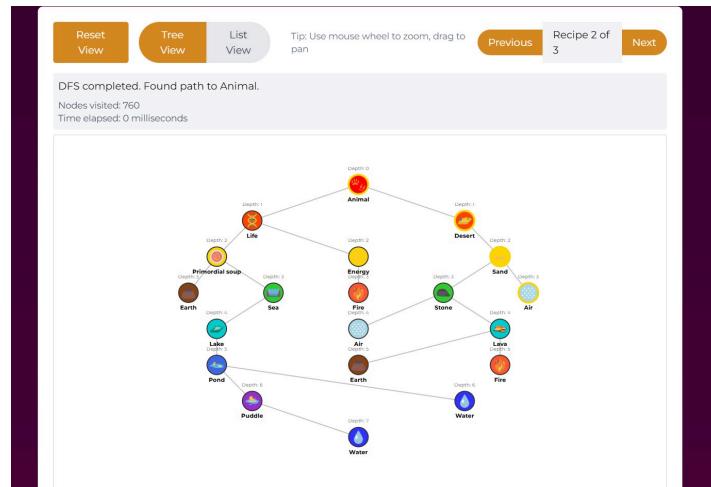


Gambar 20. Pengujian DFS Animal Single

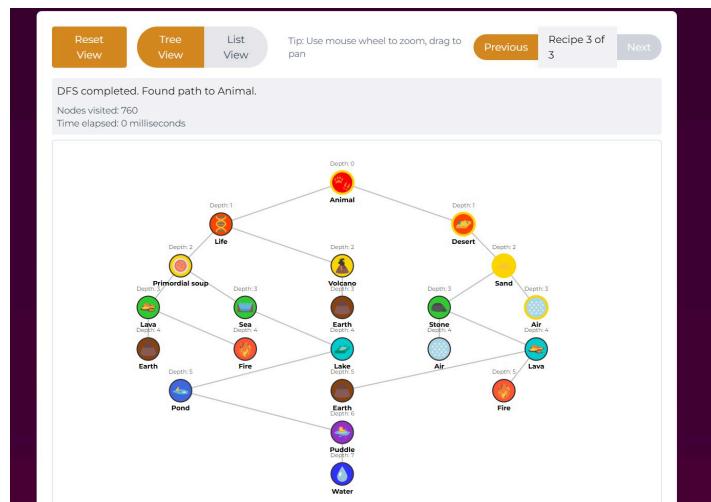
ii. Multiple



Gambar 21. Pengujian DFS Animal Multiple (1/3)

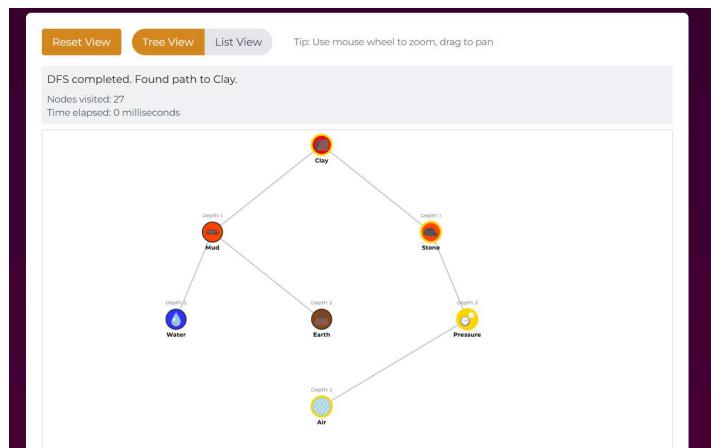


Gambar 22. Pengujian DFS Animal Multiple (2/3)



Gambar 23. Pengujian DFS Animal Multiple (3/3)

c. Elemen Clay
i. Single

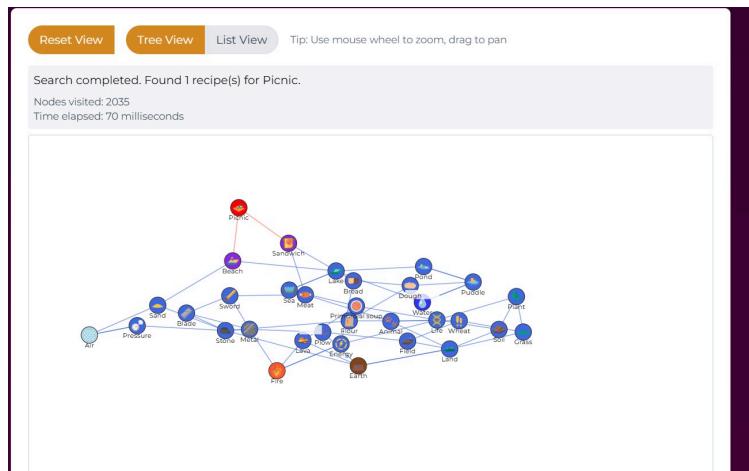


Gambar 24. Pengujian DFS Clay Single

4.3.3. Pengujian Bidirectional

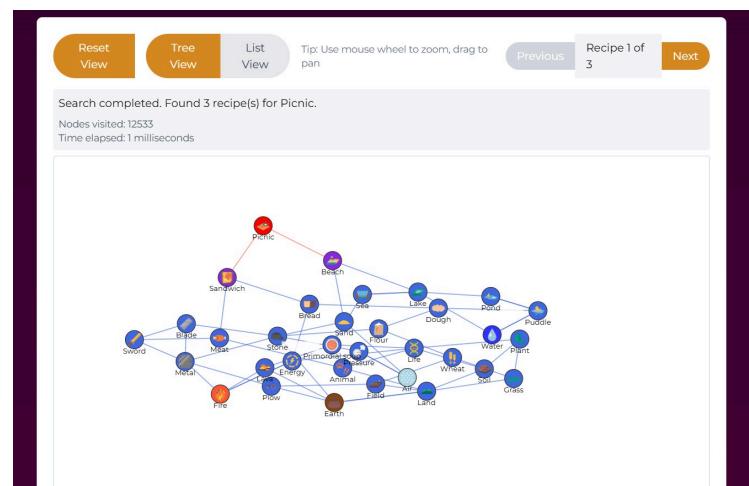
a. Elemen Picnic

i. Single

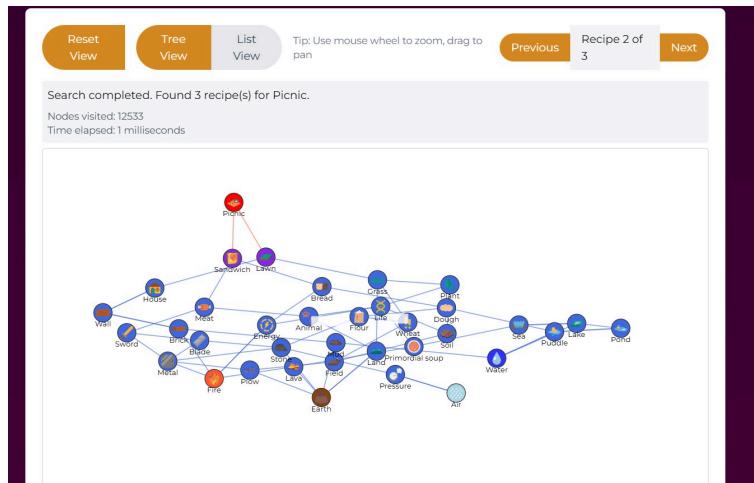


Gambar 25. Pengujian Bidirectional Picnic Single

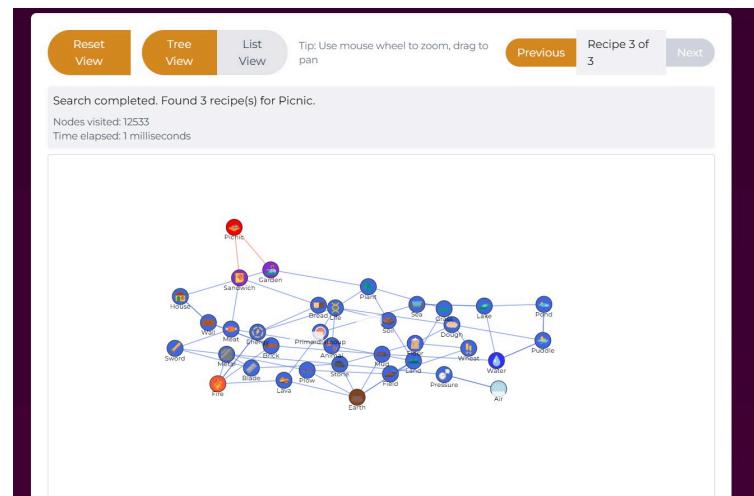
ii. Multiple



Gambar 26. Pengujian Bidirectional Picnic Multiple (1/3)

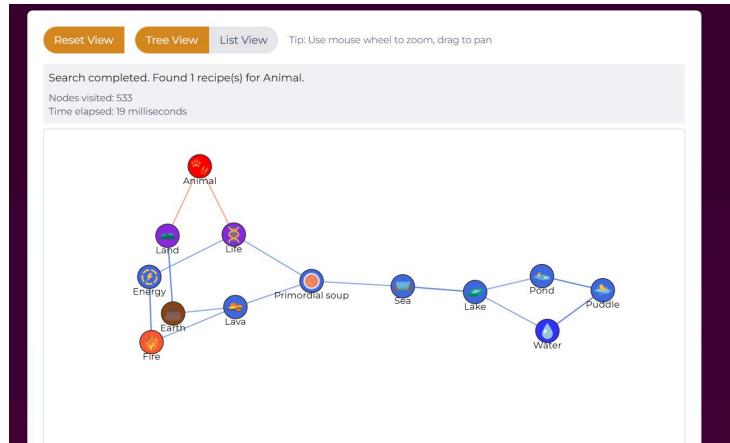


Gambar 27. Pengujian Bidirectional Picnic Multiple (2/3)



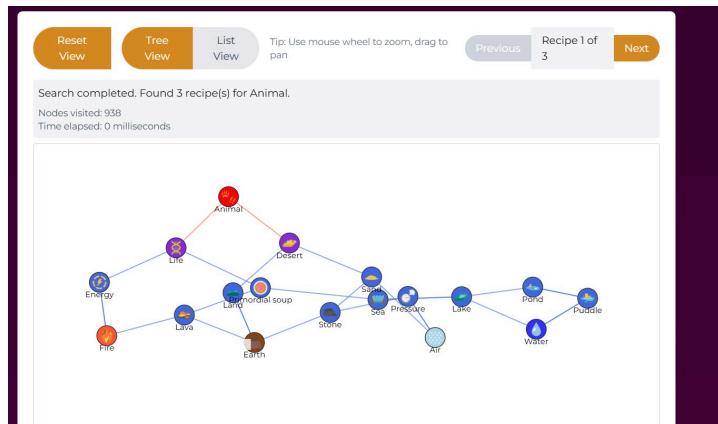
Gambar 28. Pengujian Bidirectional Picnic Multiple (3/3)

- b. Elemen Animal
 - i. Single

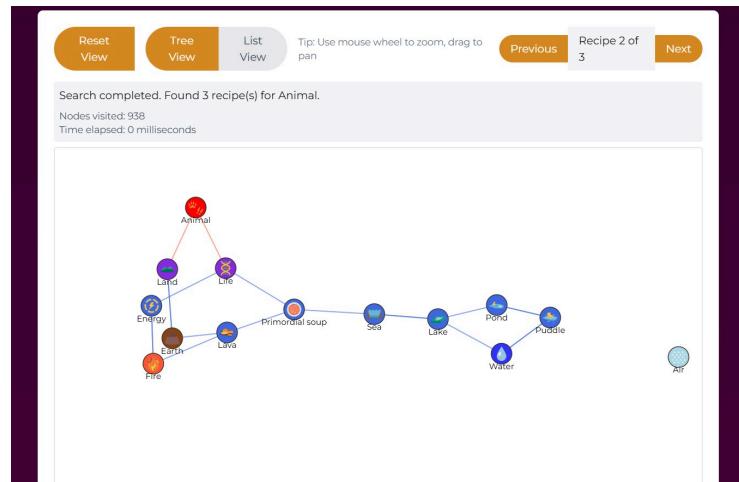


Gambar 29. Pengujian Bidirectional Animal Single

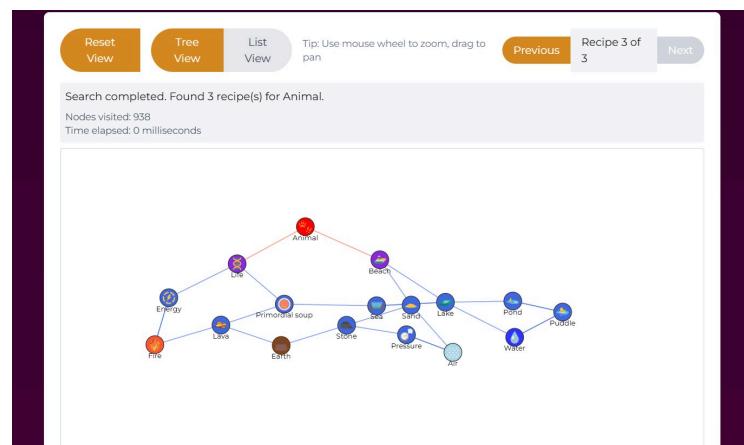
ii. Multiple



Gambar 30. Pengujian Bidirectional Animal Multiple ($\frac{1}{3}$)

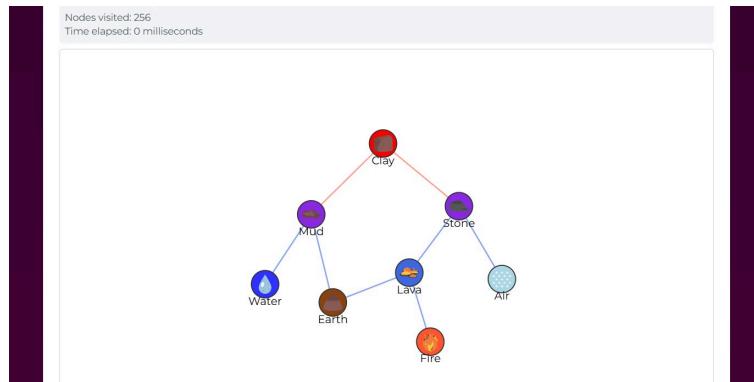


Gambar 31. Pengujian Bidirectional Animal Multiple ($\frac{2}{3}$)



Gambar 32. Pengujian Bidirectional Animal Multiple (3/3)

c. Elemen Clay
i. Single



Gambar 33. Pengujian Bidirectional Clay Single

4.4. Analisis Pengujian

Dari hasil pengujian, setiap elemen berhasil ditemukan menggunakan ketiga metode (DFS, BFS, dan Bidirectional), dengan variasi jumlah resep yang dihasilkan. Hasil multiple recipe dari tiap metode menunjukkan jalur yang unik. Waktu eksekusi untuk tiap elemen relatif singkat, karena optimasi yang dilakukan dengan multithreading dan heuristik yang membatasi eksplorasi node yang tidak relevan.

Jumlah node yang dikunjungi (node visited) menunjukkan bahwa DFS cenderung mengunjungi lebih banyak node dibanding BFS. Hal ini terjadi karena DFS harus melakukan backtracking untuk mengeksplorasi semua kemungkinan jalur, termasuk elemen yang mungkin tidak relevan sebelum menemukan solusi yang valid. Sebaliknya, BFS memiliki jumlah node visited yang lebih sedikit, karena mengeksplorasi level demi level dan memastikan solusi optimal pada graf tak berbobot.

Namun, Bidirectional search menunjukkan jumlah node visited yang lebih tinggi dibanding BFS, meskipun lebih cepat secara waktu eksekusi. Ini karena pendekatan dua arah pada Bidirectional search sebenarnya menjalankan dua pencarian BFS secara bersamaan, satu dari node asal dan satu dari node tujuan. Pada node asal, elemen akan mencoba menemukan jalan menuju titik pertemuan (meeting point), yang sering kali mengharuskan traversal lebih banyak node sebelum kedua arah bertemu. Dengan demikian, meskipun Bidirectional lebih cepat dalam menemukan jalur, ia memerlukan memori lebih besar karena harus menyimpan dua frontier traversal secara simultan.

BAB V

Penutup

5.1. Kesimpulan

Algoritma DFS, BFS, dan Bidirectional memiliki kelebihan dan kekurangannya masing-masing. Dengan DFS yang cenderung memakan memori lebih sedikit namun belum dipastikan mendapatkan path yang optimal, BFS yang memakan memori lebih banyak namun pasti mendapatkan solusi yang optimal, dan Bidirectional yang lebih cepat karena memotong pencarian dari dua arah namun memakan memori yang lebih banyak lagi dari BFS. Setelah dilakukan penyesuaian dan penambahan heuristik pada algoritmanya, pencarian resep yang dilakukan melalui website sudah cukup efisien. Hal ini juga didukung oleh multithreading yang terdapat di ketiga algoritma tersebut. Hasil dari pencarian sudah sesuai dengan constraint yang diharapkan dan tidak memerlukan waktu yang lama untuk mendapatkannya.

5.2. Saran

Setelah mengerjakan tugas ini, alangkah baiknya untuk kami tetap waras karena tugas ini lumayan susah dan memerlukan perencanaan yang baik. Selain dari itu tugas ini ternyata tidak semudah itu dan sebaiknya tidak meremehkan tugas besar apapun itu.

5.3. Refleksi

Selama penggerjaan tugas ini, kami menyadari pentingnya manajemen waktu yang baik dan ketekunan dalam mengerjakan tugas besar. Tugas ini juga mengajarkan bahwa memahami dasar-dasar teori graf dan algoritma pencarian sangat penting untuk membangun aplikasi yang efisien.

Lampiran

1. Link Repository

- https://github.com/farrelathalla/Tubes2_FE_Ahsan-geming
- https://github.com/farrelathalla/Tubes2_BE_Ahsan-geming

2. Link Video

- <https://youtu.be/zlCNGZMDTI8>

3. Tabel Kesesuaian

No	Poin	Ya	Tidak
1	Aplikasi dapat dijalankan.	V	
2	Aplikasi dapat memperoleh data recipe melalui scraping.	V	
3	Algoritma Depth First Search dan Breadth First Search dapat menemukan recipe elemen dengan benar.	V	
4	Aplikasi dapat menampilkan visualisasi recipe elemen yang dicari sesuai dengan spesifikasi.	V	
5	Aplikasi mengimplementasikan multithreading.	V	
6	Membuat laporan sesuai dengan spesifikasi.	V	
7	Membuat bonus video dan diunggah pada Youtube.	V	
8	Membuat bonus algoritma pencarian Bidirectional.	V	
9	Membuat bonus Live Update.	V	
10	Aplikasi di-containerize dengan Docker.	V	
11	Aplikasi di-deploy dan dapat diakses melalui internet.	V	

Daftar Pustaka

- GeeksforGeeks. “Bidirectional Search.” *GeeksforGeeks*, 29 March 2024,
<https://www.geeksforgeeks.org/bidirectional-search/>. Accessed 13 May 2025.
- Munir, Rinaldi. *13-BFS-DFS-(2025)-Bagian1*. 2025.
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/stima24-25.htm>,
[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).