

TUGAS BESAR 2

Pemanfaatan Algoritma BFS dan DFS dalam Pencarian Recipe pada Permainan

Little Alchemy 2

IF-2211

STRATEGI ALGORITMA

BobFirstSearch

13523039	Peter Wongsoredjo
13523086	Bob Kunanda
13523109	Haegen Quinston

Dosen Pengampu:

Dr. Nur Ulfa Maulidevi, S.T, M.Sc.

Dr. Ir. Rinaldi Munir, M.T.

Monterico Adrian, S.T, M.T.

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

DAFTAR ISI

DAFTAR ISI.....	2
BAB 1 DESKRIPSI TUGAS.....	4
Komponen Permainan.....	4
Spesifikasi Wajib.....	4
Spesifikasi Bonus.....	5
BAB 2 LANDASAN TEORI.....	6
Depth-First Search (DFS).....	6
Definisi Formal.....	6
Pseudocode.....	6
Kompleksitas.....	6
Sifat dan Keunggulan.....	7
Breadth-First Search (BFS).....	7
Definisi Formal.....	7
Pseudocode.....	7
Kompleksitas.....	8
Sifat dan Keunggulan.....	8
BAB 3 ANALISIS PEMECAHAN MASALAH.....	8
Langkah Pemecahan Masalah.....	8
Deskripsi Masalah.....	8
Penyederhanaan.....	8
Pemetaan Masalah menjadi Elemen DFS/BFS.....	9
BFS.....	9
DFS.....	10
Arsitektur Web.....	11
Ilustrasi Kasus.....	12
Memilih Elemen yang Akan Dicari.....	12
Memilih Algoritma dan Mode Pencarian.....	12
Visualisasi Recipe Tree.....	12
Statistik Pencarian.....	13
BAB 4 IMPLEMENTASI DAN PENGUJIAN.....	13
Spesifikasi Teknis Program.....	13
Tata Cara Penggunaan Program.....	14
Hasil Pengujian.....	14
Analisis Hasil Pengujian.....	16
BAB 5 KESIMPULAN.....	16
LAMPIRAN.....	17
Repository GitHub.....	17
Video.....	17
DAFTAR PUSTAKA.....	17

BAB 1 DESKRIPSI TUGAS

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 Permainan

1. Elemen Dasar

Terdapat empat elemen dasar: *Water*, *Fire*, *Earth*, dan *Air*. Keempat elemen ini dapat dikombinasikan untuk membentuk elemen turunan.

2. Elemen Turunan

Total elemen turunan yang harus ditemukan sebanyak 720 buah. Setiap elemen turunan memiliki resep (kombinasi dua elemen) yang dapat berupa elemen dasar ataupun elemen turunan itu sendiri.

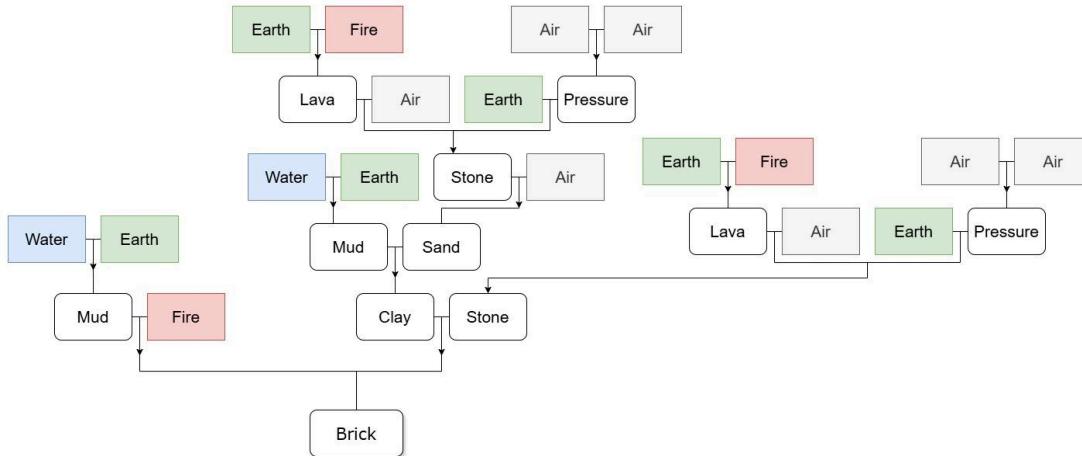
3. Mekanisme Kombinasi

Pemain menggabungkan dua elemen untuk menghasilkan elemen baru. Elemen turunan yang telah diperoleh dapat digunakan kembali dalam kombinasi selanjutnya.

Spesifikasi Wajib

- Membuat 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 terpendek** (*output* dengan rute terpendek) 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**.

- Aplikasi akan **memvisualisasikan** *recipe* yang ditemukan sebagai sebuah *tree* yang menunjukkan kombinasi elemen yang diperlukan dari elemen dasar. Tree yang dibentuk akan seperti 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.

Spesifikasi Bonus

- (**maks 5**) Membuat video tentang aplikasi BFS dan DFS pada permainan Little Alchemy 2 di Youtube. Video dibuat harus memiliki audio dan menampilkan wajah dari setiap anggota kelompok. Untuk contoh video tubes stima tahun-tahun sebelumnya dapat dilihat di Youtube dengan kata kunci “Tubes Stima”, “strategi algoritma”, “Tugas besar stima”, dll. **Semakin menarik video, maka semakin banyak poin yang diberikan.**
- (**maks 3**) Aplikasi dijalankan menggunakan **Docker** baik untuk *frontend* maupun *backend*.
- (**maks 3**) Aplikasi **di-deploy** ke aplikasi *deployment* (aplikasi *deployment* bebas) agar bisa diakses secara daring
- (**maks 3**) Menambahkan algoritma bukan hanya DFS dan BFS, tetapi juga [strategi bidirectional](#)

- **(maks 6)** Aplikasi memiliki fitur *Live Update* visualisasi *recipe* selama proses pencarian. *Tree* visualisasi akan dibangun bertahap secara *real time* sesuai dengan progress pencarian. ***Delay*** ditambahkan pada Live Update karena pencarian dapat berjalan dengan sangat cepat.

BAB 2 LANDASAN TEORI

Depth-First Search (DFS)

Definisi Formal

Diberikan graf $G = (V, E)$ dengan himpunan simpul V dan himpunan tepi E . DFS memulai dari simpul sumber $s \in V$ dan “menyelam” sedalam mungkin pada satu jalur sebelum melakukan *backtracking* untuk mengeksplorasi cabang lainnya.

Pseudocode

```

procedure DFS(u: Vertex)
    warna[u] ← “abu-abu”
    for each v ∈ Adj[u] do
        if warna[v] = “putih” then
            π[v] ← u
            DFS(v)
        warna[u] ← “hitam”

    for each u ∈ V do
        warna[u] ← “putih”
        π[u] ← NIL
    time ← 0
    DFS(s)

```

- **Warna:** {putih, abu-abu, hitam} untuk menandai status kunjungan.
- **$\pi[v]$:** prekursor, digunakan untuk merekonstruksi jalur dari setiap simpul ke sumber.

Kompleksitas

- **Waktu:** $T_{DFS}(G) = \Theta(|V| + |E|)$, karena setiap simpul diproses sekali dan setiap tepi diajukan sekali.

- **Ruang:** $S_{DFS}(G) = O(|V|)$, mencakup penyimpanan status warna, prekursor, dan *call stack* rekursif hingga kedalaman maksimum $O(|V|)$.

Sifat dan Keunggulan

- **Ketepatan Jalur:** Tidak menjamin jalur terpendek jika graf tak berbobot.
- **Completeness:** Jika G berhingga dan dapat dijangkau dari s , DFS akan mengunjungi seluruh simpul yang terhubung.
- **Aplikasi:**
 - Deteksi siklus pada graf berarah (jika menemukan tepi ke simpul abu-abu).
 - *Topological sorting* pada DAG.
 - Penyelesaian teka-teki dan *maze solving* di mana kedalaman eksplorasi prioritas.

Breadth-First Search (BFS)

Definisi Formal

BFS menelusuri graf menurut “lapisan” atau *level* jarak. Dari simpul sumber s , BFS pertama kali mengunjungi semua simpul yang berjarak satu tepi, kemudian semua simpul berjarak dua tepi, dan seterusnya.

Pseudocode

```

procedure BFS(s: Vertex)
    for each u ∈ V do
        dist[u] ← ∞
        π[u] ← NIL
    dist[s] ← 0
    buat queue Q
    enqueue(Q, s)
    while Q bukan kosong do
        u ← dequeue(Q)
        for each v ∈ Adj[u] do
            if dist[v] = ∞ then
                dist[v] ← dist[u] + 1
                π[v] ← u
                enqueue(Q, v)

```

- $\text{dist}[v]$: jarak minimum (dalam jumlah tepi) dari s ke v .
- $\pi[v]$: prekursor untuk membangun kembali jalur terpendek.

Kompleksitas

- **Waktu:** $T_{BFS}(G) = \Theta(|V| + |E|)$.
- **Ruang:** $S_{BFS}(G) = O(|V|)$, untuk array jarak, prekursor, dan queue yang pada puncaknya memuat semua simpul satu lapisan.

Sifat dan Keunggulan

- **Optimalitas Jalur:** Pada graf tak berbobot, BFS menjamin menemukan jalur terpendek (minim jumlah tepi) dari s ke setiap simpul terjangkau.
- **Completeness:** BFS akan mengunjungi seluruh simpul yang terhubung ke s .
- **Aplikasi:**
 - Pencarian jalur terpendek pada graf tak berbobot.
 - Pencarian level-order traversal pada pohon.
 - Pemodelan lapisan atau tingkatan pada jaringan sosial dan web crawling.

Aplikasi Web

Aplikasi web ini dibangun menggunakan Next.js, sebuah framework populer untuk React yang memungkinkan rendering sisi server (SSR) dan pembuatan situs statis (SSG). Struktur ini memberikan aplikasi frontend yang cepat dan skalabel.

Frontend dikembangkan menggunakan TypeScript, yang meningkatkan proses pengembangan dengan memastikan keamanan tipe data, mengurangi kesalahan runtime, dan meningkatkan kualitas kode. Next.js menyediakan berbagai utilitas, seperti API routes untuk integrasi backend, pages untuk routing, dan components untuk membuat elemen UI yang dapat digunakan kembali.

Frontend berinteraksi dengan backend melalui permintaan HTTP (melalui fetch), yang khususnya berkomunikasi dengan API routes yang didefinisikan di backend, seperti /api/tree. Frontend mengirimkan query (misalnya, jenis algoritma, mode pencarian, elemen target) dan menerima data yang mencakup pohon solusi (resep) dan statistik pencarian.

Library dan Framework Utama yang Digunakan:

- **Next.js:** Framework untuk React yang memungkinkan server-side rendering, static site generation, dan routing dinamis.

- **TypeScript**: Superset dari JavaScript yang menambahkan tipe statis, meningkatkan kualitas kode dan pengalaman pengembangan.
- **Vis Network**: Digunakan untuk memvisualisasikan pohon resep. Library ini memfasilitasi visualisasi interaktif dengan nodes dan edges, sangat cocok untuk menampilkan hasil BFS dan DFS.
- **TailwindCSS**: Framework CSS berbasis utilitas yang digunakan untuk penataan komponen. Framework ini memungkinkan sistem desain yang sangat dapat disesuaikan dan responsif.
- **Shaden UI**: Library yang meningkatkan antarmuka pengguna dengan berbagai komponen UI yang telah dibangun sebelumnya seperti tombol, label, dan kartu.

BAB 3 ANALISIS PEMECAHAN MASALAH

Langkah Pemecahan Masalah

Deskripsi Masalah

Pada game Little Alchemy 2, setiap elemen turunan dihasilkan dari penggabungan tepat dua bahan dasar atau turunan lainnya. Permasalahan utamanya adalah: “Bagaimana menemukan satu atau beberapa rangkaian langkah (resep) untuk membuat elemen target tertentu, dimulai dari empat elemen dasar?” Kompleksitas muncul karena ada ratusan puluh kombinasi yang saling berhubungan, dimana sebuah resep bisa menggunakan elemen yang juga diturunkan dari resep lain, sehingga pencarian tanpa strategi dapat dengan mudah terjebak dalam loop atau eksplorasi berlebihan.

Penyederhanaan

Untuk mengatasi tantangan ini, seluruh kumpulan resep kita representasikan sebagai graf berarah berhierarki berdasarkan tier. Setiap simpul (node) melambangkan satu elemen, sedangkan sisi (edge) berarah menghubungkan dua simpul bahan—sebagai komponen—menuju simpul target yang dihasilkan. Dalam struktur ini, empat elemen dasar (Water, Fire, Air, Earth) berperan sebagai daun, sedangkan elemen yang ingin dicapai menempati posisi akar, membentuk graf berlapis yang memudahkan penerapan metode DFS maupun BFS.

Pemetaan Masalah menjadi Elemen DFS/BFS

BFS

BFS adalah algoritma pencarian yang mengeksplorasi graf secara level per level (tier per tier), mulai dari elemen target dan bekerja mundur hingga mencapai elemen dasar. Dengan menggunakan antrian (queue), BFS memproses setiap elemen dengan mengunjungi semua tetangganya (komponen bahan) terlebih dahulu, sebelum melangkah lebih dalam ke dalam graf. Proses ini memastikan bahwa jalur resep yang ditemukan adalah yang terpendek, karena algoritma ini memprioritaskan eksplorasi elemen-elemen yang

lebih dekat terlebih dahulu. Setiap elemen yang telah dikunjungi akan ditandai dalam set visited untuk mencegah eksplorasi ulang dan menghindari siklus atau pengulangan yang tidak perlu. Penyaringan berdasarkan tingkat (tier) juga diterapkan untuk memastikan bahwa hanya resep yang relevan yang dieksplorasi, sehingga meningkatkan efisiensi pencarian.

Metode BFS yang diimplementasikan pada algoritma adalah dengan memisalkan setiap elemen yang harus ditelusuri dalam 2 komponen: List Elemen yang harus dipecah lebih lanjut, dan Rantai yang telah dibentuk sejauh ini. List elemen yang harus dipecah adalah kumpulan elemen, dari kombinasi elemen sebelumnya yang harus di dekomposisi dan diselesaikan (menjadi base elemen). Sebagai contoh: human yang merupakan komposisi dari Clay dan Life, pada penelusurannya, akan dihasilkan queue item baru untuk queue human ini, yaitu queue dengan list item di depan berupa [Clay Life], dan rantai yang terbentuk adalah rantai berupa [Human {Clay Life}]. Ketika pada depth selanjutnya node tersebut harus diproses dan dipecah, Node akan dipecah menjadi $m \times n$ Node berbeda, dengan m adalah jumlah kombinasi masing masing elemen yang bisa membentuk Clay, dan n adalah jumlah kombinasi masing masing elemen yang bisa membentuk Life, sehingga memastikan bahwa setiap kombinasi yang dapat membentuk Human, pada akhirnya tercapai melalui semua bfs ini. Sebagai ilustrasi, pada depth selanjutnya queue akan berupa [[Mud Sand Life, {Human Clay Life, Clay Mud Sand}], [Mud Stone Life, {Human Clay Life, Clay Mud Stone}]]. Setiap komponen yang berada di bagian paling depan list harus ditelusuri dahulu, dengan memecah Mud menjadi Water dan Earth, yang keduanya merupakan base element, maka dapat kita hilangkan itu dari Queue, sehingga yang tersisa nantinya adalah [Life, {Human Clay Life, Clay Mud Sand, Mud Water Earth}], dimana node ini akan diletakkan di bagian paling belakang queue. Dengan demikian memastikan bahwa penelusuran dijalankan secara berurutan per depth yang ada.

Pseudocode:

```
function BFS(target, idx, tiers, limit):
    Initialize queue as a list containing a QueueItem0:
        - Elemt: [target]
        - Chain: null
        - Depth: 0
        - Visited: empty set

    Initialize solutions as an empty list
    Initialize seenChains as an empty map
    Initialize nodesVisited as 0
    Initialize mutex (mu) for thread-safe operations
    Initialize atomic flag limitReached as 0

    for depth from 0 to 50 (or until queue is empty):
```

```

if limitReached is set to 1:
    break

Initialize nextQueue as an empty list

Initialize WaitGroup (wg) for concurrent processing

Initialize nextQueueMutex (nextQueueMu) for thread-safe queue
operations

Copy current state of queue to currentQueue

Clear queue

for each item in currentQueue:
    if limitReached is set to 1:
        continue

    Increment nodesVisited

    if item.Elems is empty:
        chainList = buildChainList(item.Chain)

        if isFullyResolved(chainList):
            key = generateChainKey(chainList)

            if key not in seenChains:
                seenChains[key] = true

                Append chainList to solutions

                if length of solutions >= limit:
                    Set limitReached to 1

                continue

    if limitReached is set to 1:
        continue

    Initialize localNextQueue as an empty list

    elem = item.Elems[0]

    rest = item.Elems[1:]

    if elem is a base element or elem is already visited:

```

```

Add a new QueueItem0 with:

    - Elems: rest

    - Chain: item.Chain

    - Depth: item.Depth

    - Visited: copy(item.Visited) to localNextQueue

else:

    recipes = idx[elem]

    for each recipe in recipes:

        c1, c2 = components of recipe

        tierOfElem = tiers[elem]

        if tiers[c1] >= tierOfElem or tiers[c2] >= tierOfElem:

            continue

        node = new ChainNode with:

            - Recipe: {Result: elem, Components: [c1, c2]}

            - Parent: item.Chain

        newElems = rest

        if c1 is not visited and c1 is not a base element:

            Append c1 to newElems

        if c2 is not visited and c2 is not a base element:

            Append c2 to newElems

        newVisited = copy(item.Visited)

        newVisited[elem] = true

        Add new QueueItem0 to localNextQueue with:

            - Elems: newElems

            - Chain: node

            - Depth: depth + 1

            - Visited: newVisited

    if localNextQueue is not empty and limitReached is not set to

```

```

1:

    Lock nextQueueMutex

    Append localNextQueue to nextQueue

    Unlock nextQueueMutex

    Wait for all goroutines to finish (using wg.Wait())

    if length of solutions >= limit:

        Return the first limit solutions and nodesVisited

    Update queue with nextQueue for the next depth level

Return solutions and nodesVisited

```

DFS

DFS adalah algoritma pencarian yang menelusuri graf dengan mengikuti satu cabang atau jalur hingga mencapai elemen dasar, kemudian kembali dan melanjutkan pencarian ke cabang lainnya. DFS diimplementasikan secara rekursif, di mana setiap elemen dieksplorasi dengan mengikuti salah satu komponen (anak) hingga kedalaman tertentu. Setelah mencapai elemen dasar (base element), algoritma ini melakukan backtracking untuk mengeksplorasi komponen lainnya. Proses ini memungkinkan DFS untuk menggali jalur resep secara mendalam, memberikan wawasan tentang semua kemungkinan resep yang dapat dibentuk, meskipun jalur tersebut mungkin lebih panjang. DFS dapat dihentikan begitu kedua komponen dalam resep adalah elemen dasar, menandakan bahwa jalur resep telah lengkap.

```

function DFSAll(target, built, limit, parent):

    Initialize nodesVisited to 1

    If target is a base element:

        Return a chain containing the target and its parent, along with
        nodesVisited

    If target is already built:

        Return null, nodesVisited

    Initialize allChains as an empty list

    Initialize seenChains as an empty map

    Initialize WaitGroup (wg) for concurrent execution

    Initialize mutex (mu) for thread-safe operations

```

```

Initialize leftChains, rightChains as empty lists of Step
Initialize leftVisited, rightVisited to 0
If target has a list of components in recipeMap:
    For each components in componentsList:
        If the number of allChains has reached the limit:
            Break out of the loop
        Assign left, right as components[0], components[1]
        If the elementMap value of target is greater than both left and
right:
            If left equals right and left is not a base element:
                leftChains, leftVisited = DFSAll(left, copy of built,
limit, target)
                Increment nodesVisited by leftVisited
                For each chain in leftChains:
                    If the number of allChains has reached the limit:
                        Break out of the loop
                    Create a new chain by appending the current Step to
the chain
                    Append the new chain to allChains
            Else:
                Increment WaitGroup by 2
                Start a goroutine to:
                    Call DFSAll for left with a copy of built, limit,
and target
                    Add the result to leftChains, and update
leftVisited
                Start a goroutine to:
                    Call DFSAll for right with a copy of built, limit,
and target
                    Add the result to rightChains, and update
rightVisited

```

```

rightVisited

    Wait for both goroutines to finish (using wg.Wait())

    For each chain in leftChains:

        For each chain in rightChains:

            If the number of allChains has reached the
            limit:

                Break out of the loop

            Create a new chain by combining the left and
            right chains

                Add the parent step to the chain

                Generate a unique signature for the new chain

                If the chain signature is not in seenChains:

                    Add the chain to allChains

                    Mark the chain signature as seen in
                    seenChains

                Set built[target] to true

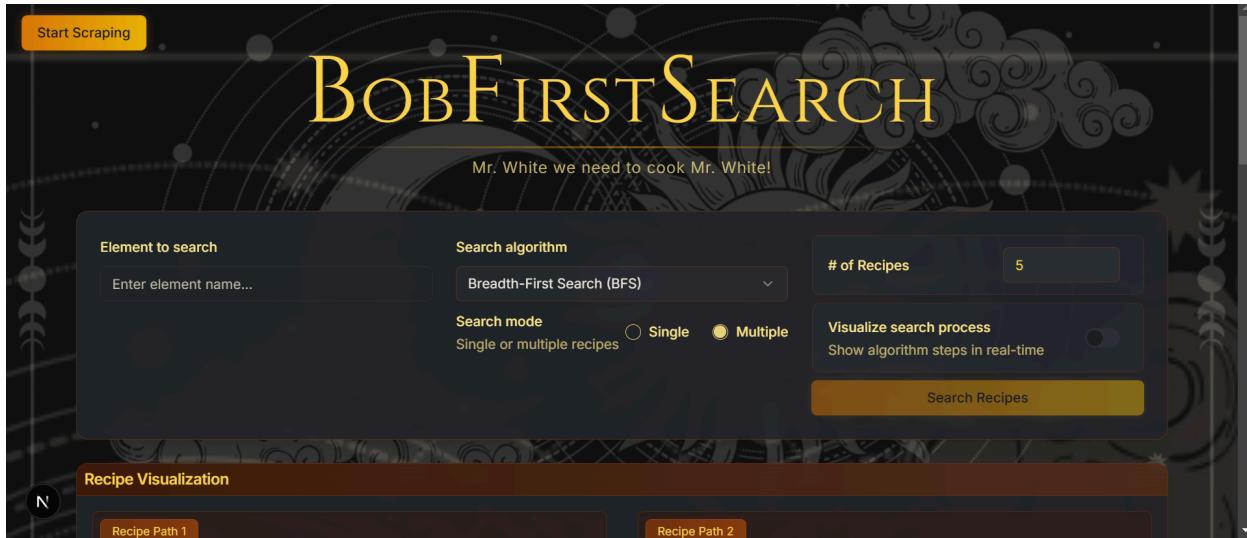
            Return allChains and nodesVisited

```

Arsitektur Web

Arsitektur web ini dirancang untuk mendukung fungsionalitas pencarian resep dalam permainan **Little Alchemy 2** menggunakan algoritma **BFS** dan **DFS**. Web ini terdiri dari beberapa komponen utama, yaitu **Main Title**, **Stats Panel**, **Search Panel**, dan **Recipe Tree**. **Stats Panel** menampilkan statistik terkait runtime, seperti waktu pencarian dan jumlah simpul yang dikunjungi. **Search Panel** memungkinkan pengguna untuk memasukkan elemen target yang ingin dicari, memilih algoritma pencarian, serta menentukan jumlah resep yang ingin ditemukan. **Recipe Tree** berfungsi untuk menghubungkan ke backend, di mana ia mengirimkan permintaan pencarian pohon resep dan menampilkan visualisasi pohon ke pengguna.

Fitur utama web ini mencakup pencarian dinamis dengan algoritma yang dapat dipilih pengguna, visualisasi hasil pencarian dalam bentuk pohon menggunakan **vis-network**, serta tampilan statistik yang memberikan wawasan tentang efisiensi pencarian. Selain itu, pengguna dapat memilih mode pencarian (jalur terpendek atau beberapa resep), dan dapat memperbesar atau memperkecil tampilan pohon resep yang dihasilkan. Web ini juga menyediakan tema gelap yang dapat diaktifkan untuk meningkatkan kenyamanan pengguna.

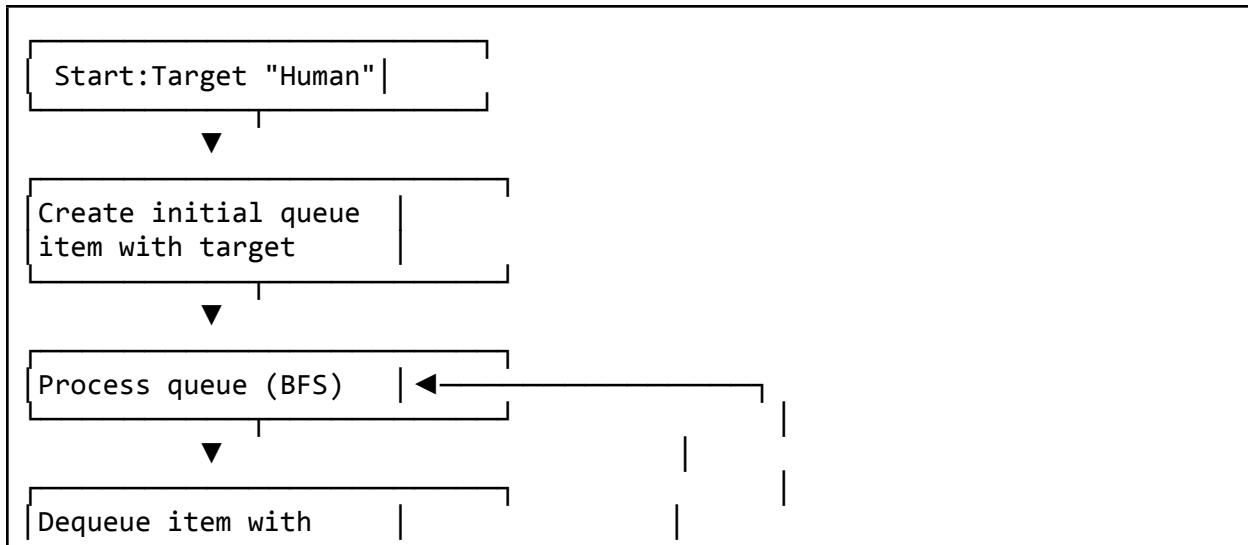


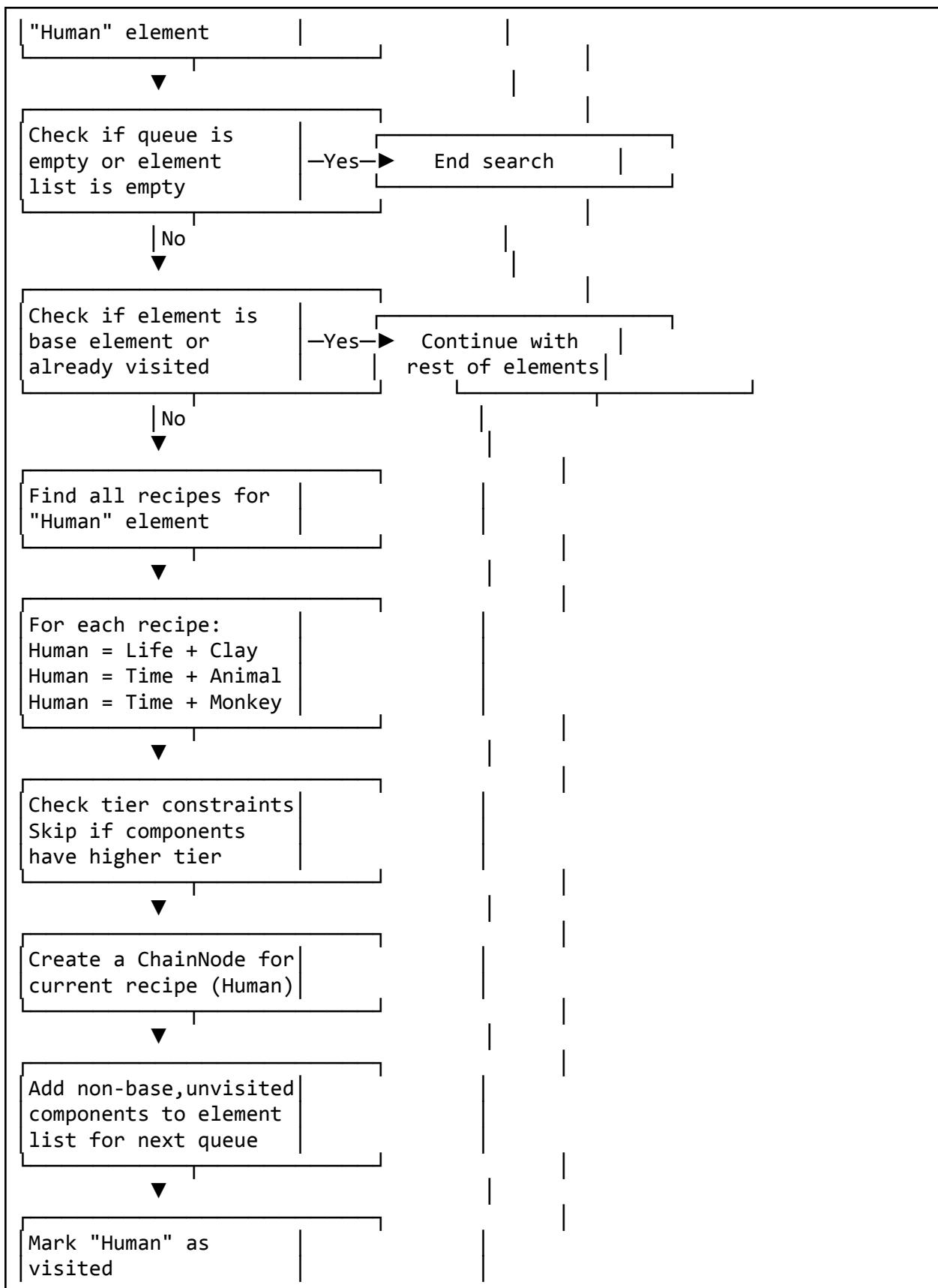
Ilustrasi Kasus

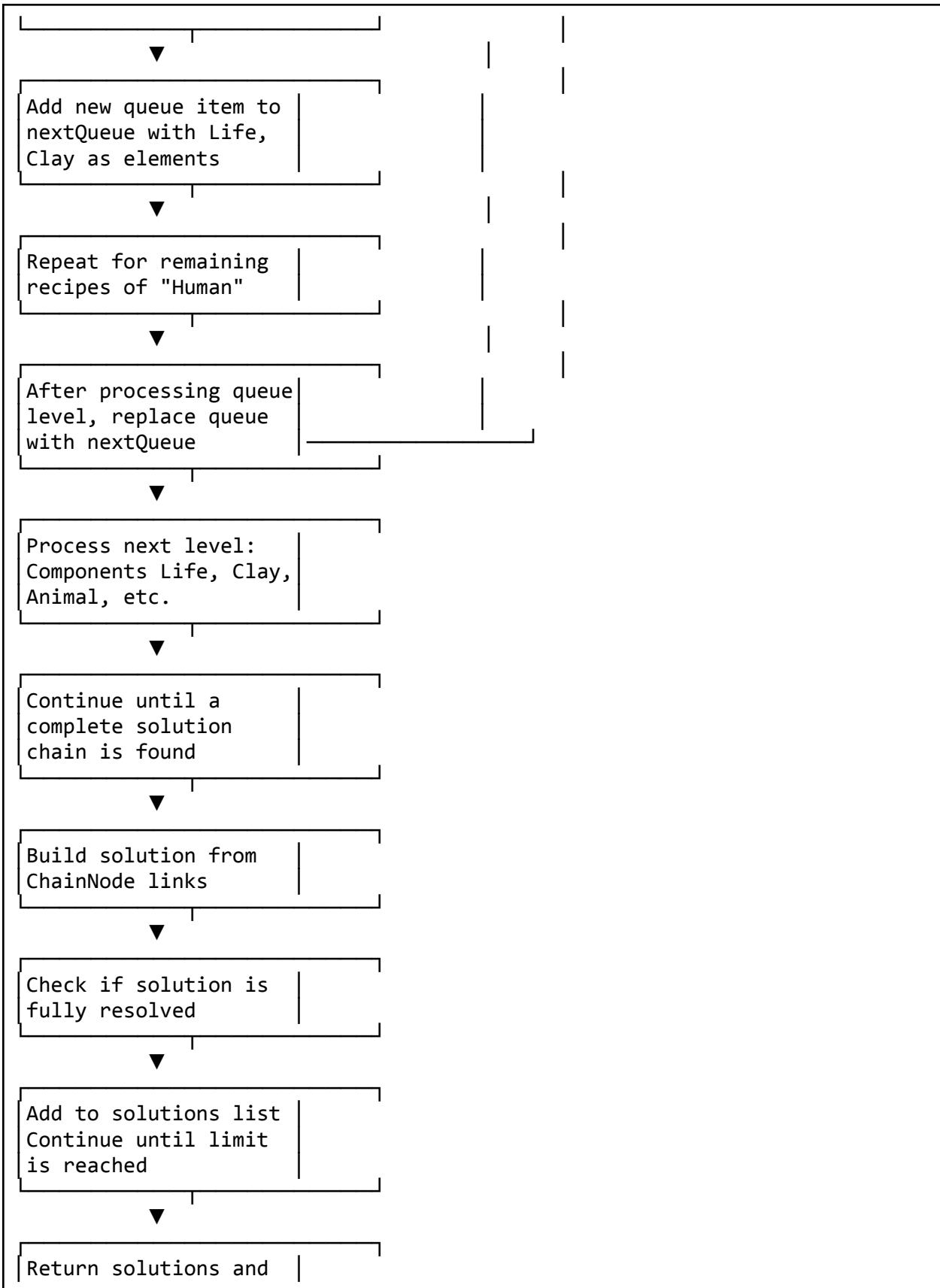
Alur dimulai dari front-end, di mana pengguna akan memasukkan elemen yang ingin dicari, memilih algoritma pencarian (BFS, DFS, atau Split BFS), serta menentukan jumlah resep yang ingin ditampilkan. Pengguna kemudian menekan tombol untuk memulai pencarian, yang memicu pengiriman permintaan HTTP ke server backend. Permintaan ini berisi parameter seperti elemen target, algoritma pencarian yang dipilih, mode pencarian (singkat atau banyak), dan jumlah resep yang diinginkan.

Begitu permintaan diterima oleh back-end, server akan mengambil parameter yang diterima dan menggunakan salah satu algoritma pencarian untuk menemukan pohon resep yang sesuai. Proses ini dilakukan dengan dua algoritma utama yang dapat dipilih pengguna, yaitu BFS (Breadth-First Search) dan DFS (Depth-First Search).

BFS Flowchart

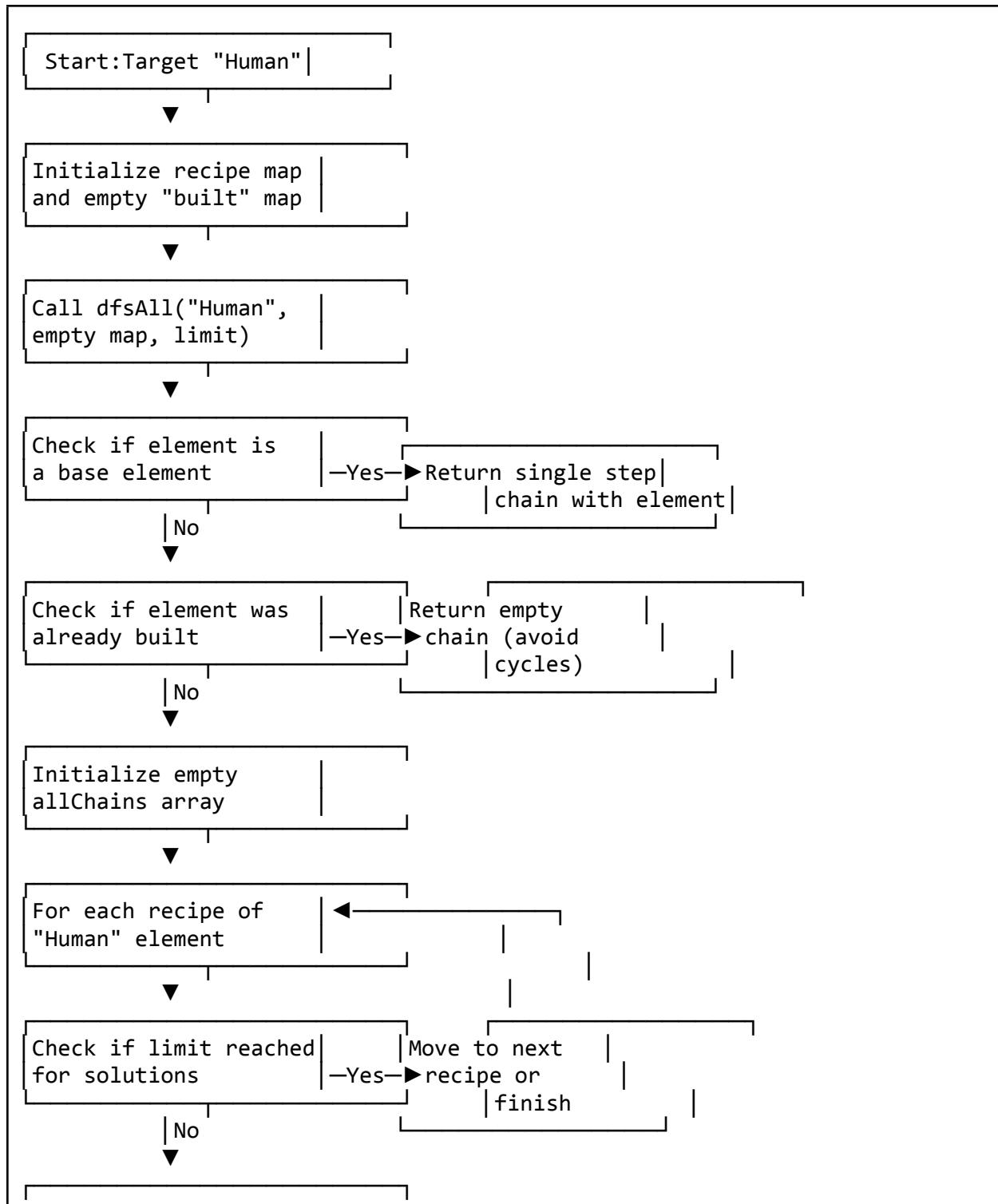


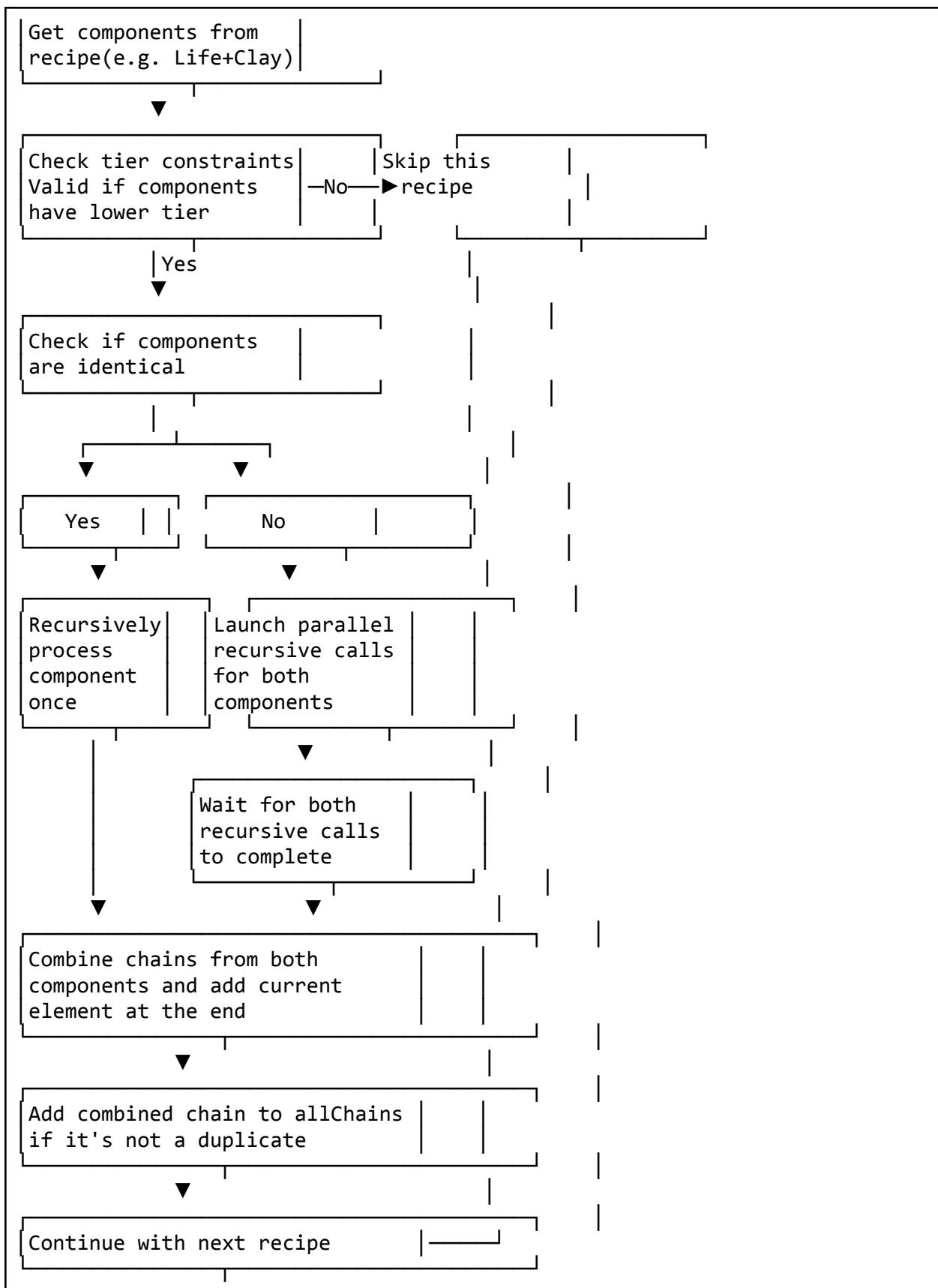


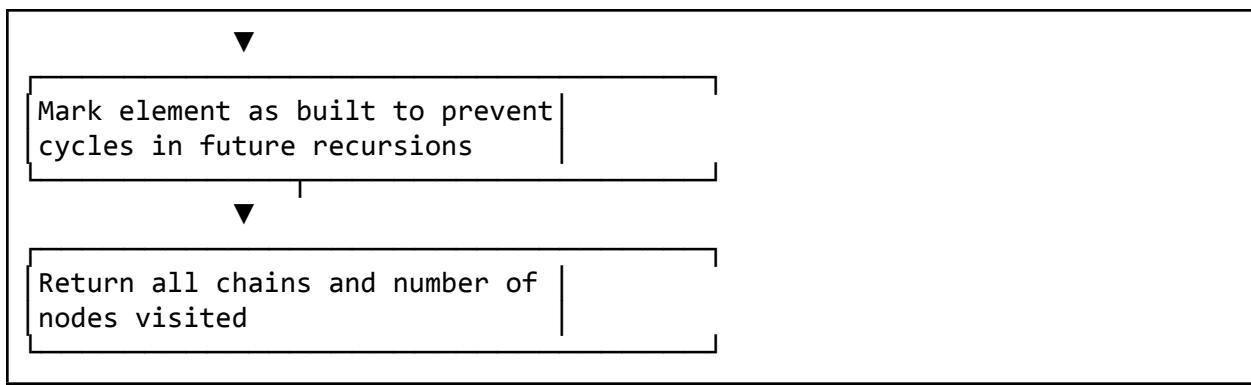




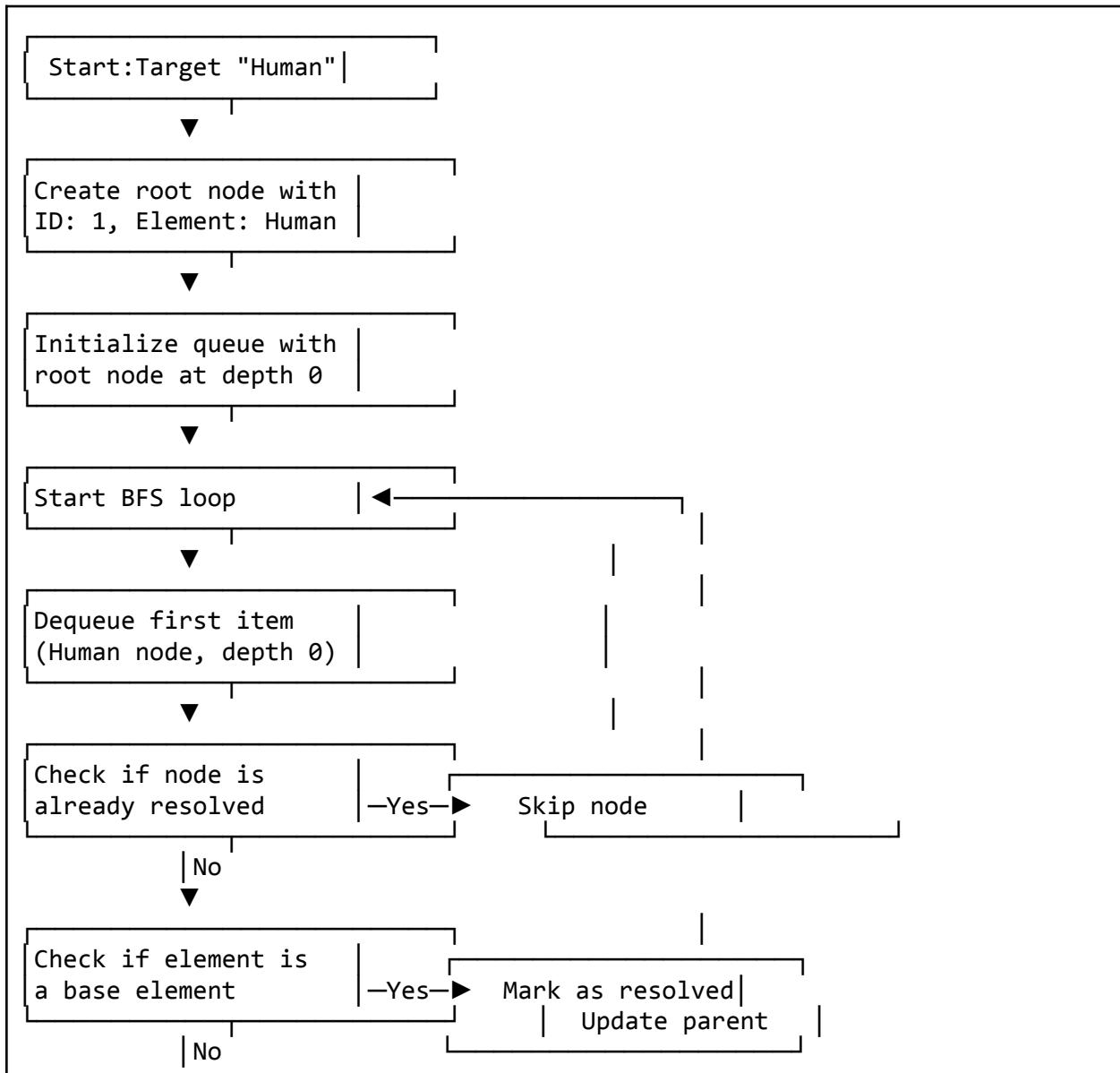
DFS Flowchart

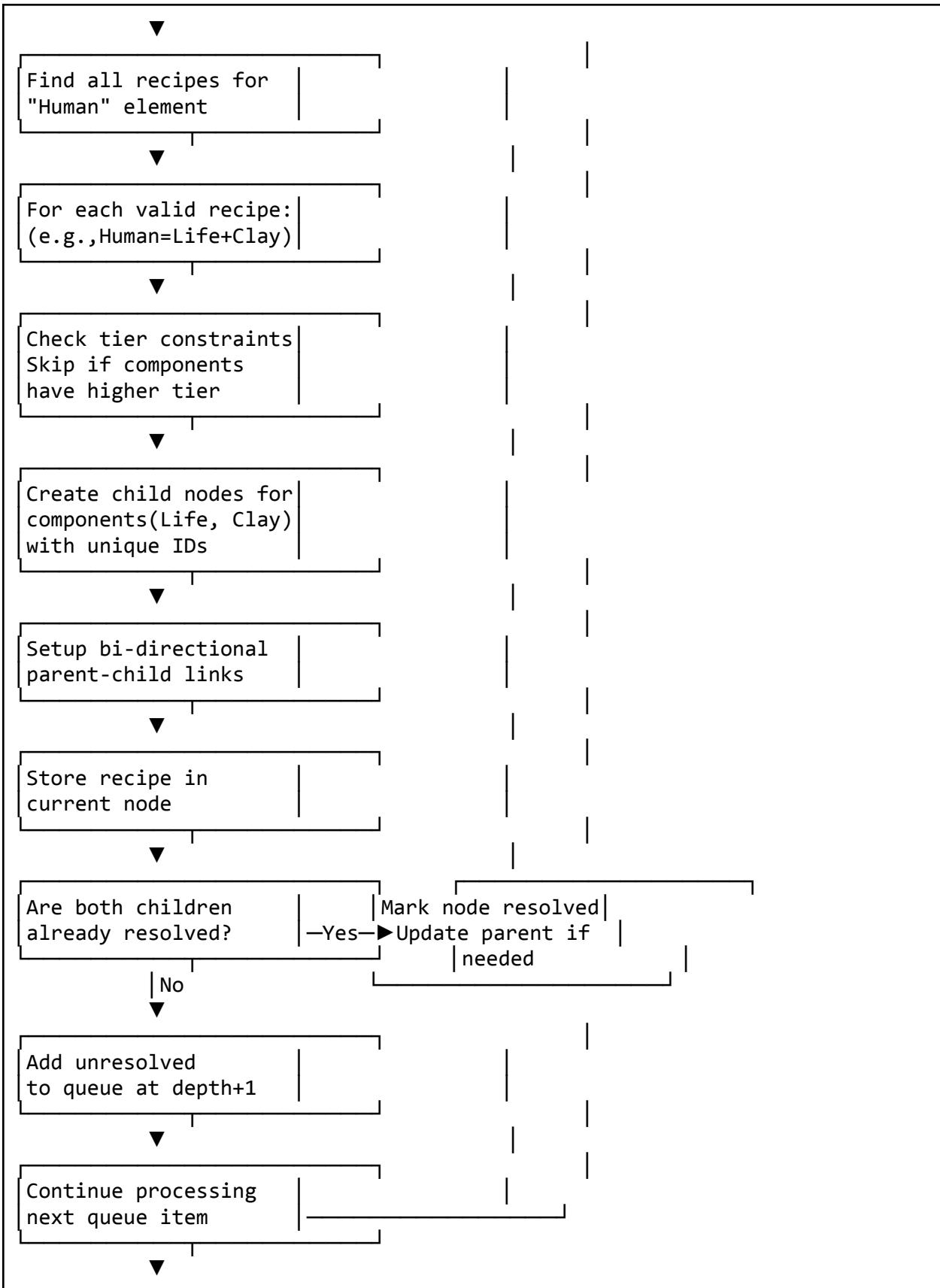


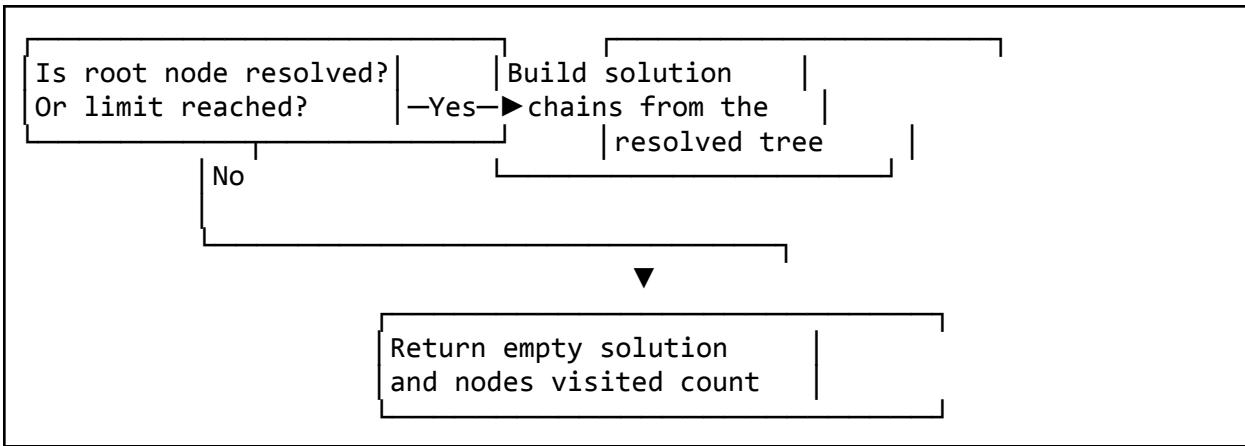




Split BFS Flowchart







Setelah algoritma BFS atau DFS selesai menjalankan pencarian, hasilnya dalam bentuk pohon resep akan disusun oleh backend. Pohon ini akan terdiri dari nodes yang merepresentasikan elemen-elemen yang ditemukan dan edges yang menunjukkan hubungan antar elemen tersebut. Server akan mengirimkan data ini dalam format JSON kepada front-end untuk divisualisasikan dalam bentuk graph.

Di front-end, RecipeTree akan menangani penerimaan dan penampilan data pohon tersebut. Menggunakan pustaka Vis Network, pohon resep ditampilkan secara interaktif di layar pengguna. Setiap node mewakili elemen, dan garis penghubung (edges) menunjukkan bagaimana elemen-elemen tersebut dapat digabungkan untuk menghasilkan elemen target. Pengguna dapat melihat langkah-langkah atau jalur yang diambil untuk mencapai elemen target tersebut, serta memperbesar atau memperkecil tampilan pohon untuk melihat detail lebih lanjut.

Selain menampilkan pohon resep, back-end juga mengirimkan informasi statistik mengenai pencarian, seperti waktu pencarian, jumlah node yang dikunjungi, dan jumlah resep yang ditemukan. Statistik ini ditampilkan pada bagian StatsPanel di front-end, memberikan pengguna wawasan mengenai seberapa efisien pencarian tersebut dilakukan.

BAB 4 IMPLEMENTASI DAN PENGUJIAN

Spesifikasi Teknis Program

Struktur Data

Recipe (utils.go)

```

type Recipe struct {
    Result     string `json:"result"`
}

```

```
Components []string `json:"components"`
}
```

Tujuan: Merepresentasikan sebuah resep dalam permainan. Setiap resep terdiri dari hasil (elemen yang dihasilkan) dan komponen-komponen (bahan yang diperlukan untuk membuat hasil tersebut).

- **Field:**

- Result: Nama elemen yang dihasilkan oleh resep.
- Components: Daftar komponen (bahan) yang dibutuhkan untuk membuat elemen hasil.

Element (utils.go)

```
type Element struct {
    Name string
    Tier int
}
```

Tujuan: Merepresentasikan elemen dalam permainan, beserta tingkat (tier) yang dimilikinya.

- **Field:**

- Name: Nama elemen.
- Tier: Tingkat (tier) elemen yang menunjukkan posisi elemen dalam hierarki elemen. Elemen dengan tier yang lebih rendah adalah elemen dasar (base elements), sedangkan elemen dengan tier yang lebih tinggi adalah elemen yang lebih kompleks.

Node & Edge (utils.go)

```
type Node struct {
    ID     int      `json:"id"`
    Label string `json:"label"`
}
```

```

        Parent int `json:"parent,omitempty"`
    }

type Edge struct {
    From int `json:"from"`
    To   int `json:"to"`
}

```

Tujuan: Struktur yang digunakan untuk merepresentasikan graf yang menggambarkan pencarian resep.

- **Node:** Merepresentasikan sebuah titik dalam graf dengan ID unik (ID), label (Label), dan referensi ke parentnya.
- **Edge:** Merepresentasikan hubungan (arah) antara dua titik dalam graf, yaitu dari node From ke node To.

GraphResponse (utils.go)

```

type GraphResponse struct {
    Nodes []Node `json:"nodes"`
    Edges []Edge `json:"edges"`
}

```

Tujuan: Struktur yang berisi seluruh graf yang akan dikirimkan ke frontend untuk visualisasi. Struktur ini memuat daftar **Node** dan **Edge** yang membentuk solusi graf.

MultipleGraphsResponse (TreeHandler.go)

```

type MultipleGraphsResponse struct {
    Graphs []GraphResponse `json:"graphs"`
    Stats  StatsResponse   `json:"stats"`
}

```

Tujuan: Struktur respons yang berisi beberapa graf solusi beserta statistik pencarian.

- Graphs: Daftar objek GraphResponse yang merepresentasikan berbagai pohon solusi.
- Stats: Objek StatsResponse yang berisi statistik tentang pencarian (waktu pencarian, jumlah node yang dikunjungi, jumlah resep yang ditemukan).

StatsResponse (TreeHandler.go)

```
type StatsResponse struct {  
    SearchTime int `json:"searchTime"  
    NodesVisited int `json:"nodesVisited"  
    RecipesFound int `json:"recipesFound"  
}
```

Tujuan: Struktur yang menyimpan statistik tentang pencarian.

- SearchTime: Waktu yang dihabiskan untuk menyelesaikan pencarian dalam milidetik.
- NodesVisited: Jumlah node yang dikunjungi selama pencarian.
- RecipesFound: Jumlah resep yang ditemukan selama pencarian.

Fungsi Utilitas dan Struktur Data

Fungsi Utilitas di utils.go

- **loadRecipes:** Memuat data resep dari file JSON yang ditentukan, mengembalikan daftar objek Recipe.
- **loadTiers:** Memuat data tier untuk elemen, mengembalikan peta yang memetakan nama elemen ke tingkatnya.
- **buildIndex:** Membangun indeks dari data resep, memetakan setiap elemen hasil ke komponen-komponennya untuk pencarian yang cepat.
- **copyMap:** Membuat salinan dari peta boolean (digunakan untuk melacak elemen yang sudah dikunjungi selama pencarian).

- **collectEdgesFromChain**: Mengubah rantai resep menjadi daftar pasangan parent-anak untuk pembangunan graf.
- **isFullyResolved**: Memeriksa apakah sebuah rantai resep sepenuhnya terselesaikan, yaitu semua komponen non-elemen dasar memiliki resep pembuatannya dalam rantai.
- **deduplicateChain**: Menghilangkan duplikasi dalam rantai resep, hanya mempertahankan resep pertama untuk setiap elemen hasil.
- **chainKey**: Membuat kunci unik untuk rantai resep, digunakan untuk mengidentifikasi dan menghilangkan duplikasi rantai.

Handler API dan Pembangunan Graf

Pembangunan Graf di BFS.go

- **buildMultipleTrees**: Membangun beberapa graf dari rantai resep, dengan elemen root sebagai akar.
- **buildTrueTree**: Membangun representasi graf yang akurat dari daftar pasangan parent-anak, memastikan hubungan antar node yang benar.

Pembangunan Graf DFS di DFS.go

- **buildTrueTreeFromDFS**: Mengubah rantai langkah-langkah DFS menjadi pohon graf, dengan penanganan khusus untuk format data DFS.

Struktur Data BFS dan BFSSplit

ElementStatus (BFSSplit.go)

```
type ElementStatus int

const (
    Pending ElementStatus = iota
    Processing
    Resolved
)
```

Tujuan: Merepresentasikan status resolusi dari sebuah elemen selama pencarian.

- Pending: Elemen belum diproses.
- Processing: Elemen sedang diproses.
- Resolved: Elemen telah terselesaikan.

ElementNode (BFSplit.go)

```
type ElementNode struct {

    ID      int
    Element string
    Status  ElementStatus
    Parent  *ElementNode
    Children []*ElementNode
    Recipe   Recipe
}
```

Tujuan: Merepresentasikan node elemen dalam pencarian BFSplit. Setiap node menyimpan ID unik, nama elemen, status, referensi ke parent dan anak-anaknya, serta resep yang digunakan.

QueueItem (BFSplit.go)

```
type QueueItem struct {

    Node  *ElementNode
    Depth int
}
```

Tujuan: Item dalam antrian BFSplit yang berisi referensi ke node elemen dan kedalaman pencarian saat ini.

Struktur Data BFS

- **ChainNode (BFS.go):** Merepresentasikan node dalam rantai resep BFS, berisi resep dan referensi ke parentnya.

- **QueueItemO (BFS.go):** Item dalam antrian BFS yang berisi daftar elemen yang akan diproses, rantai resep saat ini, kedalaman pencarian, dan peta elemen yang telah dikunjungi.

Struktur Data DFS

- **Step (DFS.go):** Merepresentasikan langkah dalam algoritma DFS, menyimpan elemen target dan parentnya.

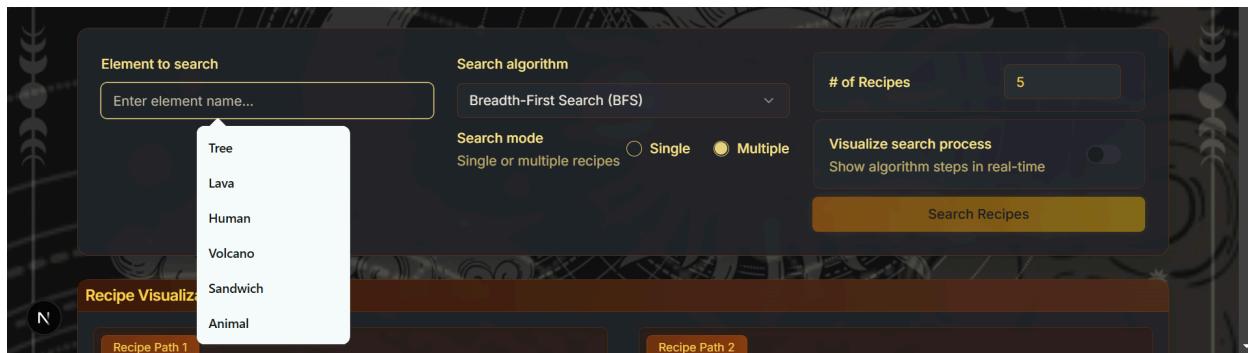
Tata Cara Penggunaan Program

Scraping



Sebelum melakukan apapun pada web, pengguna harus melakukan scraping terlebih dahulu agar elemen pada Little Alchemy 2 bisa terdefinisi pada saat searching. Ketika pengguna menekan tombol scraping, lagu *Breaking Bad Main Title Theme* dinyalakan untuk hadir menemani pengguna menunggu.

Memilih Elemen yang Akan Dicari



Pengguna dapat mengetik nama elemen pada search bar di sebelah kiri halaman. Elemen yang terdefinisi adalah elemen yang memunculkan gambar setelah diketik pada search bar.

Memilih Algoritma dan Mode Pencarian

Kedua toggle di atas merupakan opsi yang pengguna aplikasi dapat gunakan untuk memilih mode dan mode pencarian. Dalam mode multiple recipe, muncul panel yang memungkinkan pengguna untuk memilih jumlah solusi yang diinginkan (dalam range 1-50).

Visualisasi Recipe Tree



Sehabis menekan tombol ‘Search Recipes’, ditampilkan secara langsung pohon recipe yang terbentuk pada bagian ‘Recipe Visualization’. Pohon recipe ini bisa di zoom-in dan zoom-out, serta node yang terbentuk pun dapat dimainkan, dengan cara digeser kanan-kiri.

Statistik Pencarian

SEARCH RESULTS			
Search Time 52 ms	Nodes Visited 11852	Recipes Found 1	Algorithm Search Mode
			Breadth-First Search Shortest Path

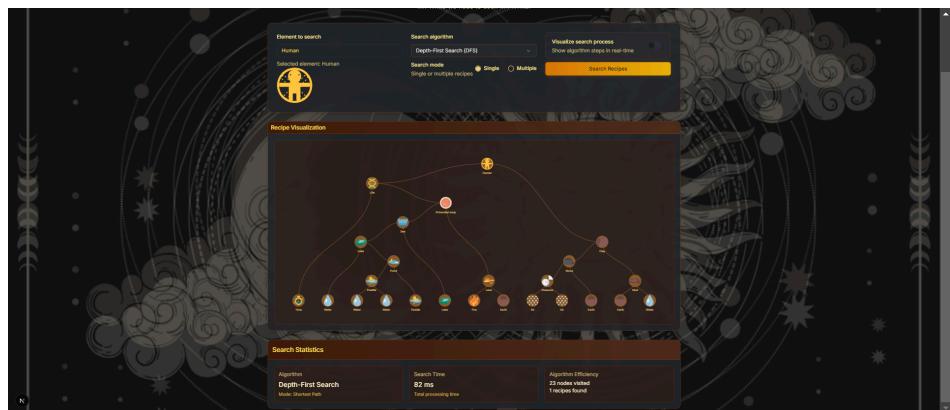
Statistik pencarian dapat ditinjau dibawah visualisasi pohon recipe tree, contohnya seperti berikut. Statistik pencarian mencakup searching time, jumlah node yang dikunjungi, jumlah recipe gabungan total, serta algoritma dan search mode yang digunakan.

Pengujian dan Hasil

Nama Pengujian	Hasil Pengujian (SS)

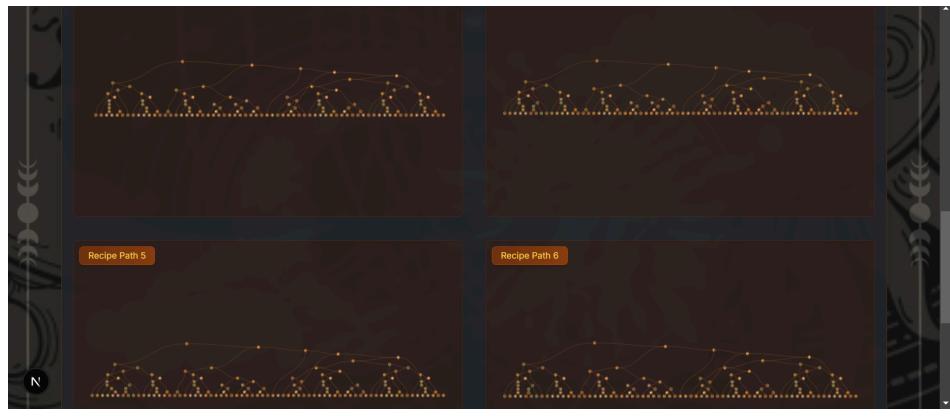
Pengujian 1

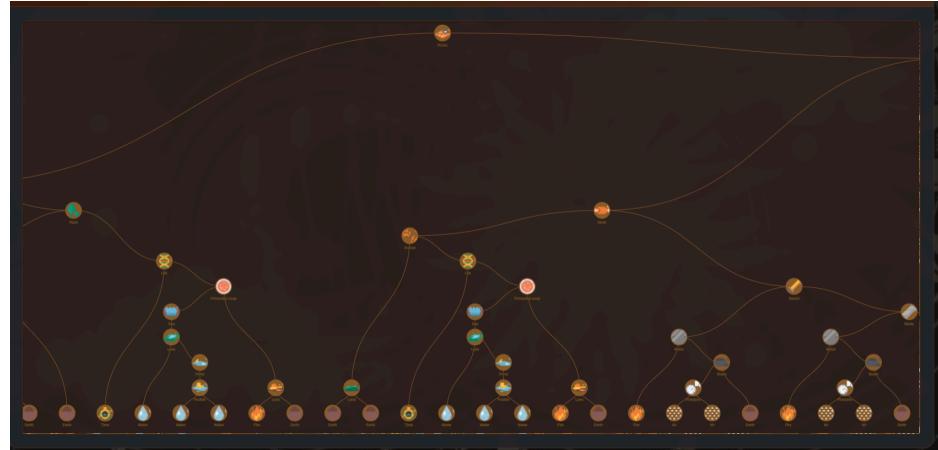
Mencari elemen Human (Tier 7) dengan algoritma DFS, keluaran single recipe.



Pengujian 2

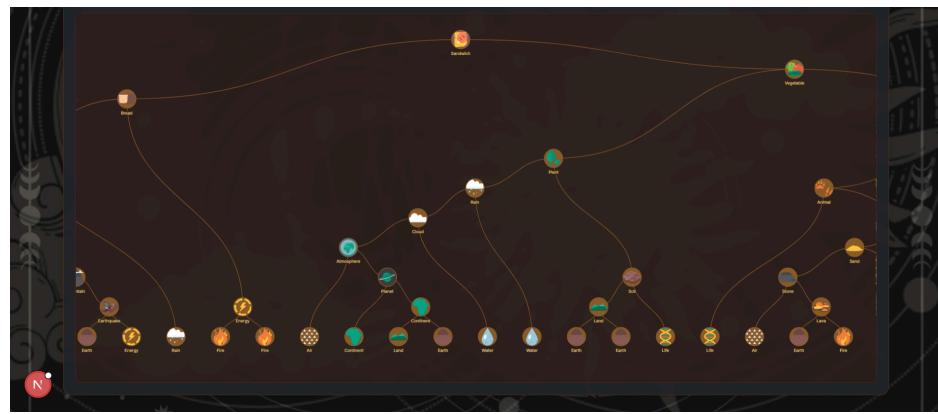
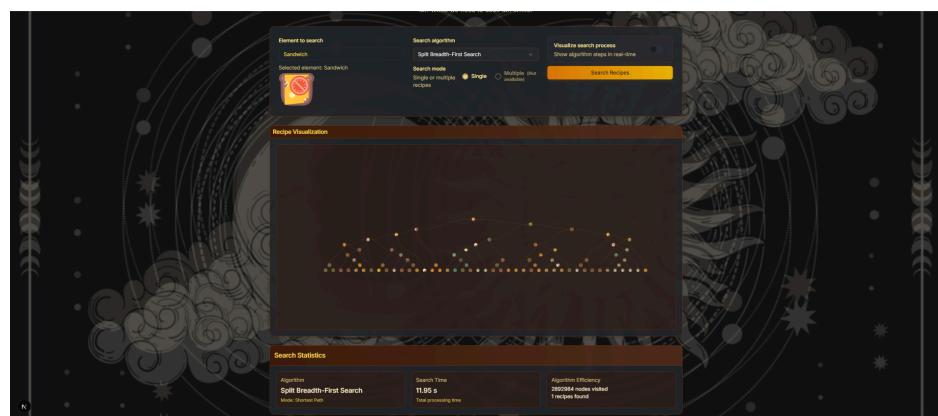
Mencari elemen Picnic (Tier 15) dengan algoritma DFS, keluaran multiple recipe (6).





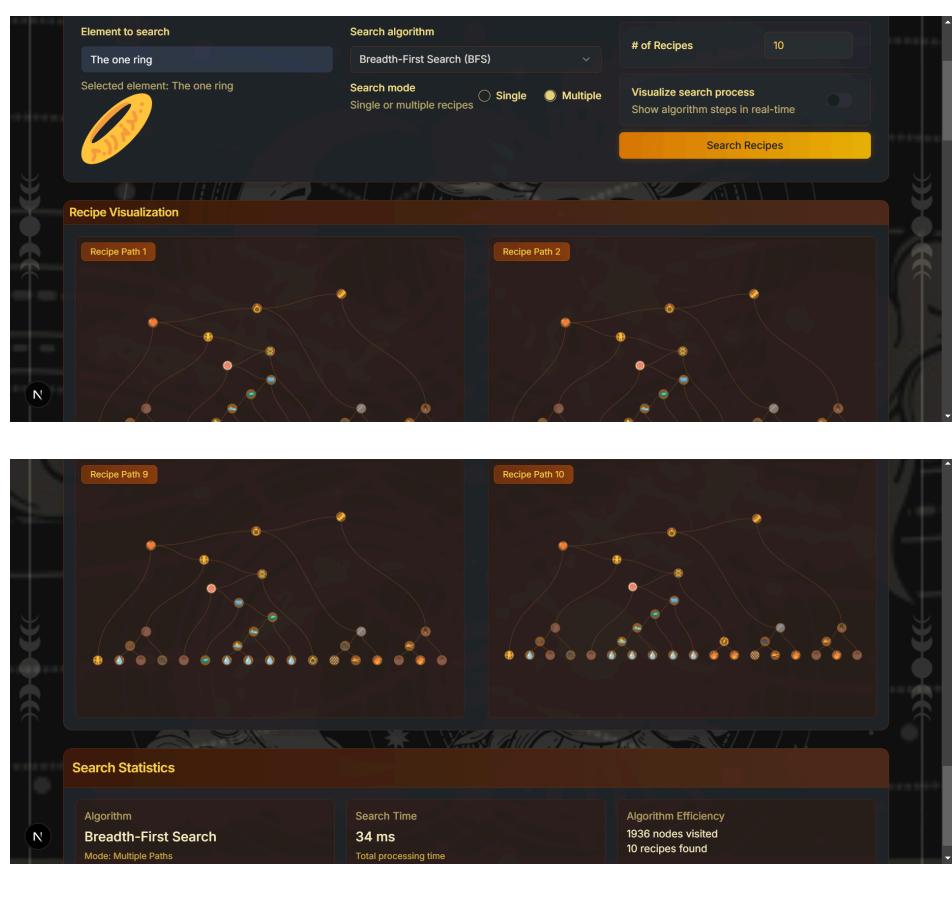
Pengujian 3

Mencari elemen Sandwich (Tier 14) dengan algoritma Split BFS, keluaran single recipe.



Pengujian 4

Mencari elemen The one ring (Tier 10) dengan algoritma BFS, keluaran multiple recipe (10).



Analisis Hasil Pengujian

Berdasarkan hasil pengujian yang telah dilakukan, dapat disimpulkan bahwa implementasi algoritma Depth-First Search (DFS) dan Breadth-First Search (BFS) berhasil diterapkan untuk pencarian elemen dalam permainan Little Alchemy 2. Pengujian ini menunjukkan bahwa algoritma DFS lebih efisien dalam menemukan solusi dibandingkan dengan BFS, terutama ketika ukuran pohon pencarian semakin besar. Hal ini disebabkan oleh sifat DFS yang menelusuri jalur secara mendalam terlebih dahulu (depth-first), yang memungkinkan pencarian dilakukan pada satu cabang sampai selesai sebelum melanjutkan ke cabang lainnya. Sebaliknya, BFS cenderung mengeksplorasi elemen per level atau tier, yang sering kali membutuhkan lebih banyak waktu untuk menemukan solusi, terutama ketika graf pencarian berkembang secara luas.

Pada pengujian dengan elemen yang berada pada Tier 14, waktu yang dibutuhkan untuk menemukan solusi tercatat cukup signifikan, yaitu 11.95 detik. Waktu eksekusi yang besar ini disebabkan oleh ukuran pohon pencarian yang berkembang pesat pada tingkat ini, menyebabkan kompleksitas waktu dan ruang menjadi sangat besar. Ini menunjukkan bahwa dengan peningkatan kedalaman dan ukuran graf, kompleksitas eksponensial mulai berpengaruh pada kinerja pencarian.

Analisis lebih lanjut menunjukkan bahwa jumlah node dari Tier 10 hingga Tier 14 bertambah secara eksponensial, mulai dari 1938 node pada Tier 10 hingga mencapai 2,892,984 node pada Tier 14.

Peningkatan yang signifikan ini menambah beban pada penggunaan memori dan waktu eksekusi, yang dapat menjadi masalah besar mengingat keterbatasan kapasitas memori pada perangkat keras yang digunakan.

Dari segi implementasi algoritma, DFS lebih unggul dalam hal efisiensi waktu pada graf yang besar, karena ia menggali lebih dalam ke dalam pohon dan mempersempit ruang pencarian. Namun, kelemahan utama dari DFS adalah potensi stack overflow atau peningkatan penggunaan memori jika pencarian terus dilakukan tanpa pembatasan yang jelas. Di sisi lain, BFS meskipun lebih memakan waktu dalam menemukan solusi, dapat lebih efektif dalam pencarian solusi terpendek, meskipun membutuhkan lebih banyak memori dan waktu pada level yang lebih tinggi.

Untuk meningkatkan efisiensi, disarankan untuk mengoptimalkan kedua algoritma ini, misalnya dengan menerapkan penyaringan berdasarkan tier atau pembatasan kedalaman pencarian, serta pembatasan jumlah solusi yang dicari. Penerapan multithreading pada BFS (seperti yang telah diimplementasikan dalam kode menggunakan goroutines) juga terbukti efektif dalam mempercepat pencarian, meskipun tetap memperhatikan batasan sumber daya sistem.

BAB 5 KESIMPULAN

Dalam pengembangan perangkat lunak, proyek ini mengimplementasikan algoritma Breadth-First Search (BFS) dan Depth-First Search (DFS) untuk pencarian resep dalam permainan Little Alchemy. Meskipun tampak sederhana, penerapan kedua algoritma ini mengandung kompleksitas tinggi dan memberikan manfaat signifikan dalam efisiensi pencarian serta visualisasi data.

Algoritma BFS memastikan pencarian dilakukan secara menyeluruh, dengan fokus pada elemen-elemen yang lebih mudah dibuat terlebih dahulu, sedangkan DFS memungkinkan eksplorasi yang lebih mendalam dari elemen dasar hingga elemen akhir. Backend program ini mengandalkan struktur data yang efisien untuk mengelola peta resep dan elemen, menjamin pencarian yang cepat dan akurat.

Kepuasan yang diperoleh dari melihat program berjalan dengan baik dan memberikan hasil yang akurat sangat besar. Implementasi algoritma ini mengajarkan pentingnya efisiensi dalam pemrograman serta bagaimana pemilihan algoritma yang tepat dapat menyelesaikan masalah kompleks dengan cara yang elegan.

Namun, saran kami terkait tugas besar ini adalah perlunya klarifikasi yang lebih baik pada spesifikasi program yang disampaikan. Informasi dalam spesifikasi awal terkesan ambigu, yang mengakibatkan kebingungannya banyak mahasiswa, termasuk kami, dalam menghadapi masalah looping pada tahap awal pengembangan. Tanpa spesifikasi yang jelas, kami akhirnya melakukan penyesuaian dan pruning yang membuat implementasi ini lebih mendekati solusi modifikasi, daripada penerapan DFS atau BFS murni. Hal ini sedikit disayangkan karena dapat menghambat pemahaman dan penerapan algoritma yang seharusnya lebih tepat.

LAMPIRAN

Repository GitHub

Video

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

DAFTAR PUSTAKA

“Little Alchemy 2”

<https://littlealchemy2.com/>

“All elements in Little Alchemy 2”

[https://little-alchemy.fandom.com/wiki/Elements_\(Little_Alchemy_2\)](https://little-alchemy.fandom.com/wiki/Elements_(Little_Alchemy_2))

“Client-Server Architecture”

https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/Client-Server_overview

“What is an API?”

<https://aws.amazon.com/what-is/api/>

“Next Documentation”

<https://nextjs.org/docs>

“React Documentation”

<https://go.dev/doc/>

“Golang Documentation”

<https://go.dev/doc/>

“Gin”

<https://gin-gonic.com/>

“goquery”

<https://github.com/PuerkitoBio/goquery>

“Effective Go Concurrency”

https://go.dev/doc/effective_go#concurrency