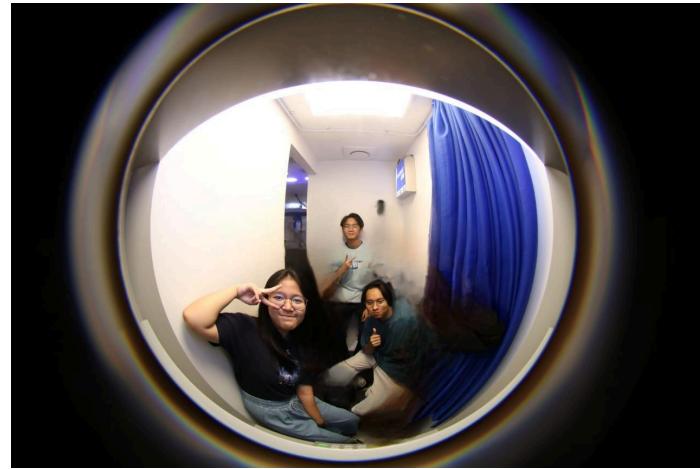


LAPORAN TUGAS BESAR II

IF2211 Strategi Algoritma

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



Disusun oleh:

“Stami”

Bertha Soliany Frandi 13523026

Albertus Christian Poandy 13523077

Ahmad Ibrahim 13523089

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG**

2025

DAFTAR ISI

DAFTAR ISI.....	2
BAB I.....	4
DESKRIPSI TUGAS.....	4
BAB II.....	4
LANDASAN TEORI.....	5
2.1 Traversal Graf.....	5
2.2 Pengembangan Website.....	6
2.3 Bahasa Pemrograman.....	8
2.3.1 Golang.....	8
2.3.2 JavaScript.....	8
BAB III.....	9
ANALISIS PEMECAHAN MASALAH.....	9
3.1 Langkah-Langkah Pemecahan Masalah.....	9
3.2 Proses Pemetaan Masalah sebagai Elemen BFS dan DFS.....	9
3.3 Proses Pengambilan Data.....	10
3.4 Fitur Fungsional dan Arsitektur Aplikasi Web.....	11
3.5 Contoh Ilustrasi Kasus.....	12
3.5.1 BFS.....	12
3.5.2 DFS.....	14
3.5.1 Bidirectional Search.....	16
BAB IV.....	19
IMPLEMENTASI DAN PENGUJIAN.....	19
4.1 Implementasi Algoritma BFS.....	19
4.1.1 Struktur Data.....	19
4.1.2 Fungsi dan Prosedur.....	19
4.2 Implementasi Algoritma DFS.....	23
4.2.1 Struktur Data.....	23
4.2.2 Fungsi dan Prosedur.....	23
4.3 Implementasi Algoritma Bidirectional.....	28
4.3.1 Struktur Data.....	28
4.3.2 Fungsi dan Prosedur.....	29
4.4 Tata Cara Penggunaan Program.....	40
4.4.1 Laman Utama.....	40
4.4.2 Laman Pencarian.....	41
4.5 Hasil Pengujian.....	43
4.5.1 Test Case 1.....	43
4.5.2 Test Case 2.....	43
4.5.3 Test Case 3.....	44
4.5.4 Test Case 4.....	44
4.5.5 Test Case 5.....	45
4.5.6 Test Case 6.....	45
4.5.7 Test Case 7.....	46

4.5.8 Test Case 8.....	46
4.6 Analisis Hasil Pengujian.....	46
BAB V.....	48
PENUTUP.....	48
5.1 Kesimpulan.....	48
5.2 Saran.....	48
5.3 Refleksi.....	48
LAMPIRAN.....	49
DAFTAR PUSTAKA.....	50

BAB I

DESKRIPSI TUGAS

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

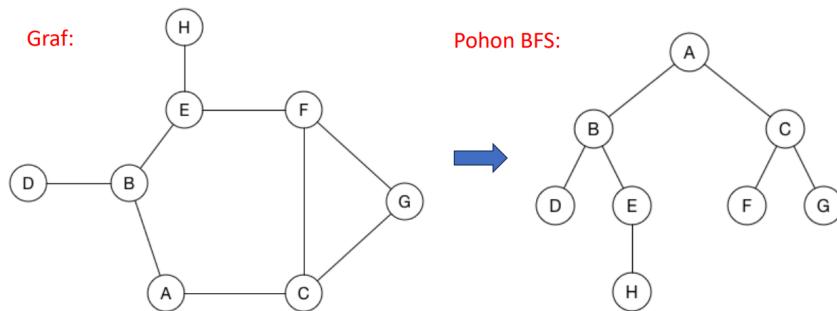
Pada Tugas Besar pertama Strategi Algoritma ini, mahasiswa diminta untuk menyelesaikan permainan Little Alchemy 2 ini dengan menggunakan strategi *Depth First Search* dan *Breadth First Search*.

BAB II

LANDASAN TEORI

2.1 Traversal Graf

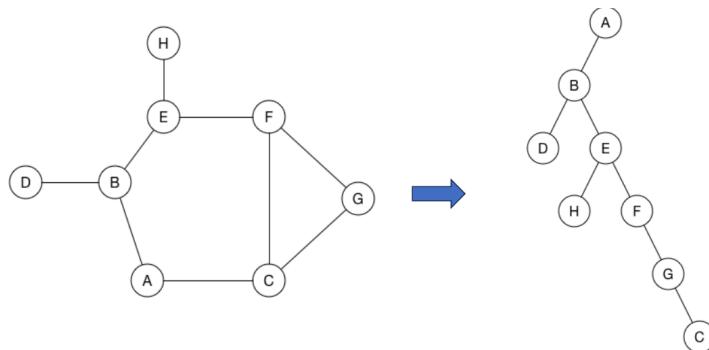
Traversal di dalam graf berarti mengunjungi simpul-simpul secara sistematik. Tujuan dari algoritma ini adalah menjelajahi semua simpul yang ada di graf beserta dengan hubungannya. Terdapat dua metode untuk algoritma traversal graf, yaitu pencarian melebar dan pencarian mendalam.



Urutan simpul-simpul yang dikunjungi secara BFS dari A → A, B, C, D, E, F, G, H

Gambar 1 Pencarian BFS

Pencarian melebar atau *Breadth First Search* (BFS) adalah algoritma pencarian yang mengunjungi simpul yang bertetangga terlebih dahulu. Contohnya terdapat simpul w. Tetangga dari simpul w ini akan terlebih dahulu dikunjungi dan setelah semua tetangga dari simpul w dikunjungi, barulah mengunjungi simpul tetangga dari simpul tetangga w, contohnya v. Kegiatan pencarian ini diulangi sampai kedalaman terdalam sudah di cek semua simpul tetangganya. Intinya, BFS mengecek per kedalaman (*level-by-level*).



Urutan simpul-simpul yang dikunjungi secara DFS dari A → A, B, D, E, H, F, G, C

Gambar 2 Pencarian DFS

Pencarian mendalam atau *Depth First Search* (DFS) adalah algoritma pencarian yang mengunjungi dari tetangga ke tetangga simpul lainnya. Maksudnya adalah DFS akan melewati simpul dari atas ke terdalam satu per satu. Begitu mencapai titik terdalam, algoritma akan melakukan *backtracking* dan mencari tetangga lain dari simpul untuk menuju ke *level* terdalam graf.

Kedua algoritma traversal graf ini dapat diaplikasikan pada *search engine* dan referensi pada makalah atau jurnal. Pada *search engine*, algoritma traversal graf diaplikasikan pada program penjelajah web (*web spider*), basis data yang menyimpan kata-kata penting pada setiap halaman web (*index*), dan pencarian berdasarkan string yang dimasukkan oleh pengguna (*query*).

Selain algoritma BFS dan DFS, terdapat algoritma pencarian lain, yaitu *Bidirectional*. *Bidirectional* adalah algoritma pencarian graf yang menemukan jalur terkecil dari simpul *start* ke simpul *goal*. Algoritma ini menjalankan dua macam pencarian, yaitu pencarian maju dan pencarian mundur. Pencarian maju yang dimaksud adalah pencarian yang dimulai dari simpul awal menuju simpul tujuan. Sedangkan pencarian mundur adalah pencarian yang dimulai dari simpul tujuan menuju simpul awal. Pencarian dengan *Bidirectional* menggantikan *single search graph* seperti BFS dan DFS dengan dua subgraf kecil dimana satunya berasal dari simpul awal dan yang lain berasal dari simpul akhir. Pencarian akan berhenti ketika kedua subgraf itu bertemu. Dalam beberapa kasus, algoritma *Bidirectional* lebih cepat dibandingkan algoritma pencarian lainnya.

2.2 Pengembangan Website

Pengembangan web adalah proses membangun dan memelihara situs web. Proses pengembangan dilakukan mulai dari desain, konten, pembuatan skrip, persiapan server hingga pengaturan keamanan jaringan. Dalam praktiknya, pengembangan *website* dibagi menjadi *frontend* dan *backend*. *Frontend* adalah antarmuka pengguna sebuah komputer atau aplikasi yang berhubungan langsung dengan pengguna. Seorang *frontend developer* bertugas untuk merancang dan membangun tampilan grafis *website* agar mudah digunakan dan dipahami oleh pengguna. *Backend* adalah bagian dari sistem yang berjalan di balik layar. *Backend developer* bertugas untuk mengelola program, server,

serta *software* situs supaya semua fiturnya berjalan dengan baik. Tugas *backend* mencakup pengelolaan API, sistem operasi, dan struktur data yang digunakan dalam aplikasi. Selain itu, terdapat pula *full-stack* yang merupakan gabungan dari *frontend* dan *backend*. Seorang *full-stack developer* bertugas untuk menciptakan situs, program software, mulai dari awal sampai akhir pembangunan.

Secara umum, proses kerja dalam pengembangan web dimulai dari pengumpulan informasi, perancangan pengembangan, perancangan antarmuka, persiapan implementasi, dan implementasi. Pada tugas ini, pengumpulan informasi dilakukan untuk memahami Tugas Besar dan permainan Little Alchemy 2. Informasi berfokus pada kombinasi elemen yang ada pada *website* yang diberikan pada dokumen spesifikasi. Rancangan pengembangan dilakukan dengan membagi tugas, penentuan struktur data hasil *scraping*, dan penyesuaianya dengan algoritma pencarian agar dapat diimplementasikan secara optimal. Perancangan antarmuka dilakukan untuk mempermudah interaksi pengguna dengan sistem. Desain harus dipastikan memiliki tempat untuk menampilkan hasil pencarian yang disetujui oleh kelompok. Pada tahap persiapan implementasi, kelompok menggunakan Docker, ESLint, dan Prettier. Docker digunakan untuk mengisolasi lingkungan pengembangan agar sesuai dengan kondisi server produksi. ESLint dan Prettier digunakan untuk menjaga kualitas dan konsistensi penulisan kode JavaScript. Setelah semua tahapan itu selesai, tahap selanjutnya, yaitu implementasi dilakukan sesuai pembagian tugas, mulai dari *scraping* data, algoritma pencarian, tampilan antarmuka web, sampai mengimplementasikan bonus seperti *live update* dan algoritma pencarian *Bidirectional*.

Setelah tahap implementasi, terdapat tahap *testing* dan *deployment*. *Testing* atau pengujian dilakukan untuk menguji fungsionalitas secara lengkap dari situs web untuk memastikan bahwa seluruh fitur berjalan dengan lancar tanpa kesalahan. Semua fitur harus diuji, terutama fitur-fitur utama dari situs web. Pengujian sebaiknya dilakukan pula untuk mengecek keresponsifan tampilan situs web dengan tujuan memberikan UI dan UX terbaik bagi pengguna. Tahap *deployment* umumnya adalah tahap akhir dalam pengembangan situs web dimana situs yang telah dikembangkan dan dites, dipindahkan ke lingkungan produksi. Lingkungan produksi yang dimaksud dapat berupa *server*, komputer pengguna, atau *cloud* yang akan digunakan *end user*.

2.3 Bahasa Pemrograman

2.3.1 Golang

Bahasa Go atau Golang adalah bahasa pemrograman yang dikembangkan oleh Google dengan tujuan menyediakan bahasa yang efisien, cepat, dan mudah dipahami. Golang bersifat *open-source* dan dikenal memiliki sintaks yang sederhana serta performa yang tinggi. Golang cocok digunakan untuk pengembangan sistem backend, cloud computing, dan aplikasi yang berjalan di sisi server.

Dalam tugas ini, Golang digunakan untuk membangun sisi *backend* dari aplikasi. Untuk mempermudah proses pengembangan dan pengelolaan routing, digunakan *framework* Gin. Gin adalah web *framework* ringan dan cepat untuk Golang yang menyediakan fitur seperti *middleware*, *JSON validation*, dan *routing* yang efisien. *Framework* ini sangat cocok digunakan dalam aplikasi RESTful karena kinerjanya yang tinggi dan kemudahannya dalam digunakan.

2.3.2 JavaScript

JavaScript adalah bahasa pemrograman yang berjalan di sisi klien dan umum digunakan dalam pengembangan web untuk menciptakan tampilan antarmuka yang dinamis dan interaktif. JavaScript bersifat *interpreter* sehingga tidak memerlukan *compiler* untuk menjalankannya.

Seiring berkembangnya teknologi, berbagai *framework* dikembangkan untuk meningkatkan kemampuan JavaScript. Dalam tugas ini, digunakan NextJS yang merupakan *framework* *open-source* React berbasis NodeJS. NextJS mendukung fitur-fitur modern seperti Server-Side Rendering (SSR), Static Site Generation (SSG), routing otomatis, dan API routes.

Untuk mempercepat dan menyederhanakan proses pembuatan komponen UI (User Interface), juga digunakan pustaka shadcn/ui. Shadcn adalah pustaka komponen yang dirancang agar kompatibel dengan Tailwind CSS dan React, serta memudahkan pengembangan antarmuka pengguna dengan desain yang konsisten dan modular.

BAB III

ANALISIS PEMECAHAN MASALAH

3.1 Langkah-Langkah Pemecahan Masalah

Permasalahan yang akan diselesaikan pada Tugas Besar kali ini adalah pencarian resep untuk membentuk suatu elemen Little Alchemy 2. Pencarian dapat berupa resep terpendek atau banyak resep yang bisa mencapai elemen tersebut. Pengguna dapat memasukkan nama elemen yang ingin dicari dan akan dibuat resep untuk elemen tersebut menggunakan algoritma pencarian *Breadth First Search* (BFS), *Depth First Search* (DFS), dan *Bidirectional*.

Resep dapat dibuat dengan algoritma pencarian dengan bantuan data dari hasil *scraping*. Hasil *scraping* ini lalu diurutkan berdasarkan tier elemen pembentuk. Mulai dari tier terkecil sampai tier tertinggi. Dari data tersebut, graf akan dibuat menggunakan algoritma pencarian dimana BFS dan *Bidirectional* menggunakan metode iteratif dan DFS menggunakan metode rekursif.

Resep dibuat berdasarkan daftar elemen dan elemen-elemen pembentuknya. Daftar ini merupakan data bertipe JSON yang disimpan setelah selesai melakukan *scraping* dari website berikut ini [https://little-alchemy.fandom.com/wiki/Elements_\(Little_Alchemy_2\)](https://little-alchemy.fandom.com/wiki/Elements_(Little_Alchemy_2)). *Scraping* adalah teknik mengambil data dari website secara otomatis. Pada program yang akan diimplementasikan *scraping* dilakukan setiap program pertama kali dijalankan.

Program diimplementasikan menggunakan *website* dengan *frontend* yang menggunakan *framework* NextJS dan shadcn/ui dengan bahasa pemrograman *Javascript* dan *Javascript Syntax Extension* (JSX), serta *backend* yang menggunakan bahasa Go.

3.2 Proses Pemetaan Masalah sebagai Elemen BFS dan DFS

Dari persoalan yang ingin dipecahkan dengan menggunakan algoritma BFS dan DFS, terdapat elemen yang bisa dipetakan, yaitu elemen-elemen pembentuk sebagai simpul. Simpul dikatakan bertetangga jika mereka merupakan elemen pembentuk dari elemen sebelumnya. Mapping elemen dari algoritma yang akan dibuat berdasarkan

pemecahan masalah tersebut adalah simpul graf untuk *start element*, *goal element*, dan *element paths*.

Algoritma BFS memanfaatkan struktur data hasil *scraping* untuk menyimpan node yang akan dieksplor selanjutnya. Pendekatan secara iteratif dilakukan dengan membentuk *queue* node dimana elemen pertama dari *queue* tersebut akan selalu merupakan elemen yang dicari semua tetangganya. Algoritma DFS memanfaatkan struktur data yang sudah diurutkan dari tier terkecil hingga tertinggi dengan menghampiri elemen pembentuk pertama agar memastikan hasil pencarian tidak terlalu dalam untuk menemukan elemen dasar. Pendekatan secara rekursif dilakukan untuk memastikan bahwa pencarian mendalam yang dilakukan dengan selalu mengecek satu tetangga terlebih dahulu dan terus menerus sebelum melakukan *backtracking*.

3.3 Proses Pengambilan Data

Proses pengambilan data, atau *scraping*, yang dilakukan untuk mengumpulkan informasi resep dan elemen kemungkinan besar melibatkan beberapa tahapan standar. Awalnya, sumber data diidentifikasi, situs web yang diidentifikasi adalah [https://little-alchemy.fandom.com/wiki/Elements_\(Little_Alchemy_2\)](https://little-alchemy.fandom.com/wiki/Elements_(Little_Alchemy_2)). Setelah itu, skrip atau program scraper akan mengakses halaman-halaman web yang relevan secara terprogram. Proses ini menggunakan permintaan HTTP untuk mengunduh konten HTML dari halaman-halaman tersebut. Konten mentah ini kemudian perlu diurai untuk mengekstrak informasi yang berguna.

Setelah konten HTML diperoleh, tahap berikutnya adalah *parsing* dan ekstraksi data. *Parser* HTML digunakan untuk menavigasi struktur dokumen, dengan memanfaatkan CSS *selector* untuk menemukan elemen-elemen HTML spesifik yang berisi nama item, bahan-bahan resep, gambar, atau informasi *tier/level*. Data yang diekstrak ini kemudian dibersihkan dari tag HTML yang tidak diperlukan dan diformat. Sebagai contoh, untuk membuat struktur Recipe (peta dari string ke array Kombinasi), scraper akan mencari pola bagaimana resep disajikan di halaman web, misalnya, mengidentifikasi dua elemen input dan satu elemen output untuk setiap resep. Demikian pula, untuk struktur Tier, informasi seperti nama elemen, tautan gambar, dan *level*-nya akan dicari dan diekstrak.

Data yang sudah bersih dan terstruktur ini kemudian disimpan dalam format yang dapat digunakan oleh aplikasi. Data hasil *scraping* ini disimpan terlebih dahulu ke dalam file .json, sehingga aplikasi tidak perlu melakukan scraping setiap kali dijalankan, melainkan cukup memuat data yang sudah ada.

3.4 Fitur Fungsional dan Arsitektur Aplikasi Web

Website dibangun dengan *frontend* yang menggunakan *framework* NextJS dan Shadcn/ui dengan bahasa JSX. Fitur yang terdapat pada website ini adalah laman utama yang berisi informasi-informasi mengenai *website* dan anggota kelompok dan laman pencarian yang digunakan untuk memilih algoritma pencarian, elemen yang ingin dibuat resepnya, banyak resep yang diinginkan, dan pilihan melihat pencarian secara *live update*.

Hasil masukkan dari pengguna pada laman pencarian akan di-*passing* ke *backend* yang menggunakan bahasa Go dan *framework* Gin. *Backend* memiliki tanggung jawab untuk menyediakan API Endpoint untuk permintaan pencarian sebuah *recipe*. Beberapa contoh *endpoint* yang disediakan adalah *endpoint* HTTP GET di *path* “search/bfs/” untuk mendapatkan hasil pencarian menggunakan metode BFS, *path* “search/dfs/” untuk mendapatkan hasil pencarian menggunakan metode DFS, dan *path* “search/bidirectional/” untuk mendapatkan hasil pencarian menggunakan metode *Bidirectional*. Seluruh *path endpoint* untuk proses pencarian yang disediakan perlu menambahkan parameter kueri *q*, *num*, dan *live*, sebagai informasi elemen apa yang ingin dicari, jumlah *recipe* yang ingin dicari, dan apakah mengaktifkan mode *live update*.

Kemudian, untuk mengelola dan menjalankan aplikasi, digunakan juga *docker*, yaitu sebuah platform yang memungkinkan aplikasi untuk di-kontainerisasi. Ini berarti aplikasi dikemas bersama dengan seluruh *dependency* yang dibutuhkan untuk berjalan, seperti kode, *runtime*, dan pustaka sistem, ke dalam sebuah unit portabel yang disebut kontainer. Alasan utama penggunaan Docker adalah untuk konsistensi lingkungan, untuk memastikan bahwa aplikasi berjalan dengan cara yang sama di *environment* mana pun.

Selanjutnya, untuk *deployment frontend* maupun *backend*, digunakan *Virtual Private Server* (VPS), yaitu menggunakan layanan DigitalOcean Droplets. VPS pada dasarnya adalah server virtual yang berjalan di infrastruktur *cloud*. Dengan VPS,

diberikan kendali penuh atas sistem operasinya, sehingga memungkinkan untuk menginstal perangkat lunak apapun yang dibutuhkan dan mengkonfigurasi server layaknya komputer fisik. Kemudian, dengan digunakannya Docker, proses menjalankan aplikasi frontend dan backend di VPS pun menjadi lebih mudah dan terstandarisasi. Hal yang perlu dilakukan hanyalah memastikan Docker terinstal di VPS, lalu menjalankan kontainer aplikasi yang sudah disiapkan sebelumnya, tanpa perlu khawatir tentang penyiapan manual dependensi aplikasi secara langsung di sistem operasi VPS.

3.5 Contoh Ilustrasi Kasus

Contoh ilustrasi kasus adalah pencarian satu resep untuk membuat elemen *clay*. Kasus diilustrasikan dengan hasil JSON yang akan dihasilkan. Terdapat id untuk setiap elemen untuk urutan elemen yang dihampiri.

3.5.1 BFS

```
{
  "message": "BFS search completed",
  "result": {
    "id": 0,
    "name": "Clay",
    "children": [
      {
        "id": 1,
        "name": "+",
        "children": [
          {
            "id": 2,
            "name": "Mud",
            "children": [
              {
                "id": 4,
                "name": "+",
                "children": [
                  {
                    "id": 5,
                    "name": "Water",
                    "children": null
                  },
                  {
                    "id": 6,
                    "name": "Earth",
                    "children": null
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
  }
}
```

```
        }
    ]
}
],
{
  "id": 3,
  "name": "Stone",
  "children": [
    {
      "id": 7,
      "name": "+",
      "children": [
        {
          "id": 8,
          "name": "Earth",
          "children": null
        },
        {
          "id": 9,
          "name": "Pressure",
          "children": [
            {
              "id": 10,
              "name": "+",
              "children": [
                {
                  "id": 11,
                  "name": "Air",
                  "children": null
                },
                {
                  "id": 12,
                  "name": "Air",
                  "children": null
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

```

    }
}
```

Dari hasil pencarian BFS di atas, dapat dilihat bahwa tetangga *brick*, yaitu elemen-elemen pembuatnya, *mud* dan *fire*, akan dihampiri terlebih dahulu dan memasukkannya ke dalam *queue* elemen yang akan dikunjungi. Setelah node *brick* selesai, algoritma akan lanjut ke elemen selanjutnya, yang dalam kasus ini adalah *mud*. Tetangga dari *mud* akan dihampiri semua terlebih dahulu, barulah pergi ke “saudaranya”, yaitu *fire* yang tetangganya akan dikunjungi pula apabila bukan merupakan elemen dasar.

3.5.2 DFS

```
{
  "message": "DFS search completed",
  "result": {
    "id": 0,
    "name": "Clay",
    "children": [
      {
        "id": 1,
        "name": "+",
        "children": [
          {
            "id": 2,
            "name": "Mud",
            "children": [
              {
                "id": 4,
                "name": "+",
                "children": [
                  {
                    "id": 5,
                    "name": "Water",
                    "children": null
                  },
                  {
                    "id": 6,
                    "name": "Earth",
                    "children": null
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
  }
}
```

```
        ],
    },
    {
        "id": 3,
        "name": "Stone",
        "children": [
            {
                "id": 7,
                "name": "+",
                "children": [
                    {
                        "id": 8,
                        "name": "Earth",
                        "children": null
                    },
                    {
                        "id": 9,
                        "name": "Pressure",
                        "children": [
                            {
                                "id": 10,
                                "name": "+",
                                "children": [
                                    {
                                        "id": 11,
                                        "name": "Air",
                                        "children": null
                                    },
                                    {
                                        "id": 12,
                                        "name": "Air",
                                        "children": null
                                    }
                                ]
                            }
                        ]
                    }
                ]
            }
        ]
    }
}
```

Dari hasil pencarian DFS di atas, dapat dilihat bahwa algoritma DFS akan selalu mengecek tetangga dari suatu elemen terlebih dahulu sampai menemukan elemen dasar, barulah algoritma akan melakukan *backtracking* untuk mencari tetangga dari elemen sebelumnya. Pencarian ini cukup dengan mengecek elemen pembentuk pertama pada hasil *scraping* dikarenakan hasilnya yang sudah diurutkan dari tier terendah ke tertinggi.

3.5.1 Bidirectional Search

```
{
  "message": "Bidirectional search completed",
  "result": {
    "Tree": {
      "id": 0,
      "name": "Clay",
      "imageSrc":
"https://static.wikia.nocookie.net/little-alchemy/images/5/55/Clay_2.svg",
      "children": [
        {
          "id": 15,
          "name": "+",
          "imageSrc": "",
          "children": [
            {
              "id": 2,
              "name": "Mud",
              "imageSrc":
"https://static.wikia.nocookie.net/little-alchemy/images/8/8b/Mud_2.svg",
              "children": [
                {
                  "id": 5,
                  "name": "+",
                  "imageSrc": "",
                  "children": [
                    {
                      "id": 3,
                      "name": "Water",
                      "imageSrc":
"https://static.wikia.nocookie.net/little-alchemy/images/f/f4/Water_2.svg",
                      "children": null
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  }
}
```

```
        },
        {
            "id": 4,
            "name": "Earth",
            "imageSrc":
"https://static.wikia.nocookie.net/little-alchemy/images/2/21/Earth_2.svg",
            "children": null
        }
    ]
}
],
},
{
    "id": 7,
    "name": "Stone",
    "imageSrc":
"https://static.wikia.nocookie.net/little-alchemy/images/e/eb/Stone_2.svg",
    "children": [
    {
        "id": 13,
        "name": "+",
        "imageSrc": "",
        "children": [
        {
            "id": 8,
            "name": "Earth",
            "imageSrc":
"https://static.wikia.nocookie.net/little-alchemy/images/2/21/Earth_2.svg",
            "children": null
        },
        {
            "id": 9,
            "name": "Pressure",
            "imageSrc":
"https://static.wikia.nocookie.net/little-alchemy/images/7/7a/Pressu
re_2.svg",
            "children": [
            {
                "id": 12,
                "name": "+",
                "imageSrc": "",
                "children": [
                {
                    "id": 10,
```

```
"name": "Air",
  "imageSrc":
"https://static.wikia.nocookie.net/little-alchemy/images/0/03/Air_2.
svg",
          "children": null
        },
      {
        "id": 11,
        "name": "Air",
        "imageSrc":
"https://static.wikia.nocookie.net/little-alchemy/images/0/03/Air_2.
svg",
          "children": null
        }
      ]
    }
  ]
}
]
}
]
}
]
}
],
{
  "NodeCount": 9,
  "TimeTaken": 38382313
}
}
```

Dari hasil pencarian Bidirectional di atas, sebenarnya kurang dapat terlihat bagaimana pencarian terjadi, tetapi hasil tersebut lebih mirip BFS daripada DFS, hal tersebut diakibatkan Bidirectional Search yang digunakan menggunakan BFS untuk kedua sisi.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

4.1 Implementasi Algoritma BFS

4.1.1 Struktur Data

Tabel 4.1.1.1 Tabel Struktur Data BFS

Struktur Data	Penjelasan
TreeNode	Struktur data rekursif sebagai representasi <i>tree</i> , yang dibentuk menggunakan <i>struct</i> pada Go. Struktur ini terdiri dari id, name, imageSrc, dan children yaitu <i>array of TreeNode</i> .
Combination	Struktur data yang digunakan sebagai representasi kombinasi dari dua elemen berbeda
Recipe	Struktur data yang dibentuk dengan struct. Merupakan sebuah map dengan <i>key</i> berupa string dan <i>value</i> berupa <i>array of Combination</i> . Struktur data ini digunakan sebagai informasi kumpulan kombinasi elemen yang mungkin untuk menghasilkan elemen lainnya.

4.1.2 Fungsi dan Prosedur

1. Fungsi SingleRecipeBFS(recipe *scraper.Recipe, start string, liveUpdate bool, wsManager *socket.ClientManager) → TreeNode

Fungsi untuk menjalankan proses pencarian satu buah resep terpendek untuk suatu elemen menggunakan algoritma BFS.

```
func SingleRecipeBFS(recipe *scraper.Recipe, tier *scraper.Tier,
start string, liveUpdate bool, wsManager *socket.ClientManager)
scraper.SearchResult {
    startTime := time.Now()
    nodeCount := 0
    id := 0
    root := scraper.TreeNode{Name: start, Id: id, ImageSrc:
(*tier)[start].ImageSrc}
    nodeCount++
    if liveUpdate {
        wsManager.BroadcastNode(root)
    }
}
```

```

queue := []*scraper.TreeNode{&root}
for len(queue) > 0 {

    currNode := queue[0]
    queue = queue[1:]

    if scraper.IsBaseElement(currNode.Name) {
        continue
    }
    combinations := (*recipe)[currNode.Name]
    next := combinations[0]
    first, second := next.First(), next.Second()
    id++
    node := &scraper.TreeNode{Name: "+" , Id: id}
    id++
    node.Children = []scraper.TreeNode{
        {Name: first, Id: id, ImageSrc:
        (*tier)[first].ImageSrc},
        {Name: second, Id: id + 1, ImageSrc:
        (*tier)[second].ImageSrc},
    }
    id++
    currNode.Children = append(currNode.Children, *node)
    nodeCount += 2
    if liveUpdate {
        time.Sleep(300 * time.Millisecond)
        wsManager.BroadcastNode(root)
    }
    queue = append(queue, &node.Children[0],
    &node.Children[1])
}

duration := time.Since(startTime)
return scraper.SearchResult{Tree: root, NodeCount: nodeCount,
TimeTaken: duration.Nanoseconds()}
}

```

2. Fungsi MultipleRecipeBFS(recipe *scraper.Recipe, start string, numRecipe int, liveUpdate bool, wsManager *socket.ClientManager) → TreeNode

Fungsi untuk menjalankan proses pencarian beberapa buah resep untuk suatu elemen menggunakan algoritma BFS. Untuk mengoptimasi pencarian, dilakukan *multithreading* menggunakan *Go Routine*.

```
func MultipleRecipeBFS(recipe *scraper.Recipe, tier
```

```

*scraper.Tier, start string, numRecipe int, liveUpdate bool,
wsManager *socket.ClientManager) scraper.SearchResult {
    startTime := time.Now()
    nodeCount := 0
    id := 0
    // Buat node root untuk elemen target
    root := scraperTreeNode{Name: start, Id: id, ImageSrc:
        (*tier)[start].ImageSrc}
    nodeCount++
    if liveUpdate {
        wsManager.BroadcastNode(root)
    }

    queue := []*scraperTreeNode{&root} // tambahkan root ke
queue

    var mutex sync.Mutex
    var wg sync.WaitGroup
    parent := map[int]*scraperTreeNode{}
    numPath := map[int]int{}
    done := false

    for len(queue) > 0 {

        currentQueue := []*scraperTreeNode{}

        for _, node := range queue {

            // jalankan go routine untuk setiap node pada queue
saat ini
            wg.Add(1)
            go func(currNode *scraperTreeNode) {
                defer wg.Done()
                if scraper.IsBaseElement(currNode.Name) {
                    return
                }
                combinations := (*recipe)[currNode.Name]
                for i, combination := range combinations {
                    mutex.Lock()
                    currId := id
                    id += 3
                    mutex.Unlock()
                    first, second := combination.First(),
combination.Second()
                    node := &scraperTreeNode{Name: "+" + currId + 1}
                    node.Children = []scraperTreeNode{
                        combination
                    }
                    currentQueue = append(currentQueue, node)
                }
            }()
        }
    }
    wg.Wait()
    return scraperSearchResult{Root: root, NumPath: numPath}
}

```

```

                {Name: first, Id: currId + 2, ImageSrc:
(*tier)[first].ImageSrc},
                {Name: second, Id: currId + 3, ImageSrc:
(*tier)[second].ImageSrc},
            }

        mutex.Lock()
        node.InitParAndNum(currNode, &parent,
&numPath)
        if done && i > 0 {
            mutex.Unlock()
            break
        }
        currNode.Children = append(currNode.Children,
*node)
        currentQueue = append(currentQueue,
&node.Children[0], &node.Children[1])
        if i > 0 {
            num := currNode.CountNumRecipe(&parent,
&numPath)
            if num >= numRecipe {
                done = true
            }
        }
        mutex.Unlock()

        if liveUpdate {
            mutex.Lock()
            time.Sleep(300 * time.Millisecond) //  

Tambahkan delay 300ms
            wsManager.BroadcastNode(root)
            mutex.Unlock()
        }
    }
}(node)
}
wg.Wait()

mutex.Lock()
nodeCount += len(currentQueue)
mutex.Unlock()

queue = currentQueue
}

duration := time.Since(startTime)
return scraper.SearchResult{Tree: root, NodeCount: nodeCount,

```

```
TimeTaken: duration.Nanoseconds()
}
```

4.2 Implementasi Algoritma DFS

4.2.1 Struktur Data

Tabel 4.1.2.1 Tabel Struktur Data DFS

Struktur Data	Penjelasan
Graf (TreeNode)	Struktur data yang dibentuk dengan menggunakan struct. Terdiri dari id, name, dan <i>array of</i> TreeNode.
Combination	Struktur data yang dibentuk dengan menggunakan struct. Merupakan <i>array</i> bertipe string dengan ukuran 2.
Recipe	Struktur data yang dibentuk dengan struct. Merupakan sebuah map dengan <i>key</i> berupa string dan <i>array of</i> Combination
SingleHelperParams	Struktur data yang dibentuk dengan menggunakan struct. Terdiri dari root yang merupakan *TreeNode. Id yang merupakan *int, liveUpdate yang merupakan bool, dan WsManager yang merupakan *ClientManager
MultipleHelperParams	Struktur data yang dibentuk dengan menggunakan struct. Terdiri dari root yang merupakan *TreeNode. id yang merupakan *int, count yang merupakan *int, mutex yang merupakan *Mutex, Wg yang merupakan *WaitGroup, LiveUpdate yang merupakan bool, dan WsManager yang merupakan *ClientManager

4.2.2 Fungsi dan Prosedur

1. Fungsi SingleRecipeDFS(recipe *scraper.Recipe, start string, liveUpdate bool, wsManager *socket.ClientManager) → TreeNode

Fungsi yang digunakan untuk mendefinisikan DFS terlebih dahulu sebelum masuk ke bagian rekursif. Fungsi ini mendefinisikan parameter-parameter yang dibutuhkan oleh fungsi *helper*.

```

func SingleRecipeDFS(recipe *scraper.Recipe, tier *scraper.Tier,
start string, liveUpdate bool, wsManager *socket.ClientManager)
scraper.SearchResult {
    startTime := time.Now()
    nodeCount := 0
    id := 0
    root := scraperTreeNode{Name: start, Id: id, ImageSrc:
(*tier)[start].ImageSrc}
    nodeCount++
    if liveUpdate {
        wsManager.BroadcastNode(root)
    }
    params := &SingleHelperParams{
        Root:      &root,
        Id:        &id,
        NodeCount: &nodeCount,
        LiveUpdate: liveUpdate,
        WsManager:  wsManager,
    }
    SingleDFSHelper(recipe, tier, start, params, &root)
    duration := time.Since(startTime)
    return scraper.SearchResult{Tree: root, NodeCount: nodeCount,
TimeTaken: duration.Nanoseconds()}
}

```

2. Prosedur SingleDFSHelper(recipe *scraper.Recipe, start string, params *SingleHelperParams, currNode *scraperTreeNode)

Prosedur untuk melakukan rekursif sehingga mendapatkan graf hasil pencarian. Prosedur memastikan untuk melakukan pencarian mendalam dengan benar dengan melakukan rekursif untuk node pertama terlebih dahulu sebelum ke node kedua.

```

func SingleDFSHelper(recipe *scraper.Recipe, tier *scraper.Tier,
start string, params *SingleHelperParams, currNode
*scraperTreeNode) {

    root := params.Root
    liveUpdate := params.LiveUpdate
    wsManager := params.WsManager
    id := params.Id

    if scraper.IsBaseElement(start) {
        return
    }
}

```

```

    }

    combinations := (*recipe)[start]

    next := combinations[0]
    first, second := next.First(), next.Second()

    (*id)++
    node := &scraper.TreeNode{Name: "+" , Id: (*id)}
    (*id)++
    node.Children = []scraper.TreeNode{
        {Name: first, Id: (*id), ImageSrc:
        (*tier)[first].ImageSrc},
        {Name: second, Id: (*id) + 1, ImageSrc:
        (*tier)[second].ImageSrc},
    }
    (*id)++
    currNode.Children = append(currNode.Children, *node)
    (*params.NodeCount) += 2

    if liveUpdate {
        time.Sleep(300 * time.Millisecond)
        wsManager.BroadcastNode(*root)
    }

    SingleDFSHelper(recipe, tier, first, params,
&node.Children[0])
    SingleDFSHelper(recipe, tier, second, params,
&node.Children[1])
}

```

3. Fungsi MultipleRecipeDFS(recipe *scraper.Recipe, start string, numRecipe int, liveUpdate bool, wsManager *socket.ClientManager) → TreeNode

Fungsi yang mengimplementasi algoritma DFS untuk mencari banyak resep bagi satu elemen. Menggunakan mutex dan waitgroup untuk mengimplementasikan *multithreading*.

```

func MultipleRecipeDFS(recipe *scraper.Recipe, tier
*scraper.Tier, start string, numRecipe int, liveUpdate bool,
wsManager *socket.ClientManager) scraper.SearchResult {
    startTime := time.Now()
    id := 0
    done := false
    var mutex sync.Mutex

```

```

var wg sync.WaitGroup
var nodeCounter int32 = 1

root := scraper.TreeNode{Name: start, Id: id, ImageSrc:
(*tier)[start].ImageSrc}

if liveUpdate {
    wsManager.BroadcastNode(root)
}

wg.Add(1)
params := &MultHelperParams{
    Root:      &root,
    Id:        &id,
    Parent:    &map[int]*scraper.TreeNode{},
    NumPath:   &map[int]int{},
    Done:      &done,
    NodeCount: &nodeCounter,
    Mutex:     &mutex,
    Wg:        &wg,
    LiveUpdate: liveUpdate,
    WsManager: wsManager,
}
go MultipleRecipeHelper(recipe, tier, start, numRecipe,
params, &root)
wg.Wait()
duration := time.Since(startTime)
finalNodeCount := int(atomic.LoadInt32(&nodeCounter))
return scraper.SearchResult{Tree: root, NodeCount:
finalNodeCount, TimeTaken: duration.Nanoseconds()}
}

```

4. Prosedur `MultipleRecipeHelper(recipe *scraper.Recipe, name string, numRecipe int, params *MultHelperParams, currNode *scraper.TreeNode)`

Prosedur yang membantu fungsi `MultipleRecipeDFS(...)` untuk melakukan rekursif agar pencarian dapat mengimplementasikan pencarian mendalam dengan menggunakan go routine.

```

func MultipleRecipeHelper(recipe *scraper.Recipe, tier
*scraper.Tier, name string, numRecipe int, params
*MultHelperParams, currNode *scraper.TreeNode) {
    wg := params.Wg
    mutex := params.Mutex

```

```

    mutex.Lock()
    parent := params.Parent
    numPath := params.NumPath
    done := params.Done
    liveUpdate := params.LiveUpdate
    wsManager := params.WsManager
    root := params.Root
    id := params.Id
    mutex.Unlock()

    defer wg.Done()

    if scraper.IsBaseElement(name) {
        return
    }
    combinations := (*recipe)[name]

    for i, combination := range combinations {

        mutex.Lock()
        currId := (*id)
        (*id) += 3
        mutex.Unlock()

        first, second := combination.First(),
        combination.Second()
        node := &scraper.TreeNode{Name: "+", Id: currId + 1}
        node.Children = []scraper.TreeNode{
            {Name: first, Id: currId + 2, ImageSrc:
                (*tier)[first].ImageSrc},
            {Name: second, Id: currId + 3, ImageSrc:
                (*tier)[second].ImageSrc},
        }

        mutex.Lock()
        node.InitParAndNum(currNode, parent, numPath)
        if (*done) && i > 0 {
            mutex.Unlock()
            break
        }
        currNode.Children = append(currNode.Children, *node)
        atomic.AddInt32(params.NodeCount, 2)
        if i > 0 {
            num := currNode.CountNumRecipe(parent, numPath)
            if num >= numRecipe {
                (*done) = true
            }
        }
    }
}

```

```

    }
    mutex.Unlock()
    wg.Add(1)
    go MultipleRecipeHelper(recipe, tier, first, numRecipe,
params, &node.Children[0])
    wg.Add(1)
    go MultipleRecipeHelper(recipe, tier, second, numRecipe,
params, &node.Children[1])

    if liveUpdate {
        mutex.Lock()
        time.Sleep(300 * time.Millisecond)
        wsManager.BroadcastNode(*root)
        mutex.Unlock()
    }

}
}

```

4.3 Implementasi Algoritma *Bidirectional*

4.3.1 Struktur Data

Tabel 4.1.3.1 Tabel Struktur Data *Bidirectional*

Struktur Data	Penjelasan
scraper.Treenode	Struktur data fundamental yang merepresentasikan sebuah <i>node</i> dalam pohon hasil pencarian. Terdiri dari Id (integer unik), Name (nama elemen), ImageSrc (<i>path</i> ke gambar elemen), dan Children (array dari scraperTreeNode yang merupakan anak-anak dari <i>node</i> tersebut). Node dengan Name = "+" merepresentasikan sebuah kombinasi resep.
scraper.Recipe	Sebuah <i>map</i> dengan <i>key</i> berupa <i>string</i> (nama elemen hasil) dan <i>value</i> berupa <i>array</i> dari scraper.Combination. Ini menyimpan semua kemungkinan resep untuk membuat sebuah elemen.
scraper.Combination	Struktur data yang merepresentasikan satu kombinasi resep, biasanya terdiri dari dua elemen (bahan). Memiliki metode seperti First() dan Second() untuk mengakses bahan-bahannya.
scraper.Tier	Sebuah <i>map</i> dengan <i>key</i> berupa <i>string</i> (nama elemen)

	dan <i>value</i> berupa <code>scraper.TierInfo</code> (yang berisi informasi <i>tier/level</i> elemen dan <i>path</i> gambar). Digunakan untuk menentukan elemen dasar dan mengurutkan <i>meeting nodes</i> .
<code>map[string]bool</code>	Digunakan secara ekstensif untuk melacak <i>node</i> yang telah dikunjungi (visitedFwd, visitedBwd, collectedMeetingNodes, pathVisited) dalam proses BFS dari kedua arah dan saat rekonstruksi jalur. <i>Key</i> adalah nama elemen (<i>string</i>), dan <i>value</i> adalah <i>boolean</i> yang menandakan status kunjungan.
<code>[]string</code>	Digunakan sebagai antrian (currentFwdQueue, nextFwdQueue, currentBwdQueue, nextBwdQueue) dalam implementasi BFS dari kedua arah (pencarian dari elemen dasar ke target, dan dari target ke elemen dasar). Juga digunakan untuk menyimpan orderedMeetingNodes.
<code>scraper.SearchResult</code>	Struktur data yang mengemas hasil akhir pencarian. Terdiri dari Tree (struktur <code>scraperTreeNode</code> hasil), NodeCount (jumlah <i>node</i> dalam pohon), dan TimeTaken (waktu eksekusi pencarian dalam nanosekon).

4.3.2 Fungsi dan Prosedur

1. Fungsi `BidirectionalSearch(recipe *scraper.Recipe, tierMap *scraper.Tier, startElementName string, numRecipesToFind int) → scraper.SearchResult`

Fungsi utama yang memproses keseluruhan proses pencarian dua arah untuk elemen `startElementName`. Fungsi ini menangani kasus-kasus awal (seperti elemen dasar atau elemen tanpa resep), kemudian untuk setiap resep yang valid dari `startElementName`, ia akan memanggil `solveSingleElementBidirectionally` untuk masing-masing bahan dalam resep tersebut. Hasil dari pemanggilan tersebut kemudian digabungkan untuk membentuk pohon hasil akhir. Fungsi ini juga menghitung total *node* dan waktu eksekusi.

```
func BidirectionalSearch(
    recipe *scraper.Recipe,
    tierMap *scraper.Tier,
```

```

        startElementName string,
        numRecipesToFind int,
    ) scraper.SearchResult {
        startTime := time.Now()
        globalNodeIdCounter = 0

        rootNode := scraper.TreeNode{Id: globalNodeIdCounter,
Name: startElementName, ImageSrc:
(*tierMap)[startElementName].ImageSrc}
        globalNodeIdCounter++

        if scraper.IsBaseElement(startElementName) {
            duration := time.Since(startTime)
            nodeCount := countNodesInTree(&rootNode)
            return scraper.SearchResult{Tree: rootNode,
NodeCount: nodeCount, TimeTaken: duration.Nanoseconds()}
        }

        initialCombinations, exists := (*recipe)[startElementName]
        if !exists || len(initialCombinations) == 0 {
            rootNode.Name = fmt.Sprintf("No recipes for %s",
startElementName)
            duration := time.Since(startTime)
            nodeCount := countNodesInTree(&rootNode)
            return scraper.SearchResult{Tree: rootNode,
NodeCount: nodeCount, TimeTaken: duration.Nanoseconds()}
        }

        numActualRecipes := len(initialCombinations)
        if numRecipesToFind > 0 && numRecipesToFind <
numActualRecipes {
            numActualRecipes = numRecipesToFind
        }
        if numRecipesToFind < 1 {
            numActualRecipes = 1
            if len(initialCombinations) == 0 {
                numActualRecipes = 0
            }
        }

        for i := 0; i < numActualRecipes; i++ {
            combination := initialCombinations[i]
            ing1 := combination.First()
            ing2 := combination.Second()

            var treeIngredient1, treeIngredient2
scraper.TreeNode

```

```

        if ing1 != "" {
            treeIngredient1 =
solveSingleElementBidirectionally(ing1, recipe, tierMap)
        }
        if ing2 != "" {
            treeIngredient2 =
solveSingleElementBidirectionally(ing2, recipe, tierMap)
        }

        globalNodeIdCounter++
        plusNode := scraper.TreeNode{
            Id:    globalNodeIdCounter,
            Name: "+",
        }

        isError1 := false
        // ... (Error checking logic for treeIngredient1)
        isError2 := false
        // ... (Error checking logic for treeIngredient2)

        if !isError1 {
            plusNode.Children = append(plusNode.Children,
treeIngredient1)
        }
        if !isError2 {
            plusNode.Children = append(plusNode.Children,
treeIngredient2)
        }

        if len(plusNode.Children) > 0 {
            rootNode.Children = append(rootNode.Children,
plusNode)
        }
    }

    nodeCount := countNodesInTree(&rootNode)
    duration := time.Since(startTime)
    return scraper.SearchResult{Tree: rootNode, NodeCount:
nodeCount, TimeTaken: duration.Nanoseconds()}
}

```

2. Fungsi `solveSingleElementBidirectionally(targetElementName string, recipe *scraper.Recipe, tierMap *scraper.Tier) → scraper.TreeNode`

Fungsi inti dari algoritma pencarian dua arah untuk satu elemen target. Fungsi ini melakukan dua proses BFS secara bersamaan: satu dari elemen-elemen dasar (`baseElementsList`) maju ke arah elemen target, dan satu lagi dari elemen target mundur ke arah elemen dasar. Ketika kedua pencarian bertemu di satu atau lebih *node* (disebut `meetingNodes`), pencarian berhenti. Kemudian, jalur dari elemen target ke *meeting node* terpilih dan jalur dari *meeting node* terpilih ke elemen dasar direkonstruksi menggunakan `reconstructSinglePathRecursive`. Jika tidak ada *meeting node* yang ditemukan, fungsi ini akan kembali menggunakan `reconstructSinglePathRecursive` standar dari elemen target ke elemen dasar.

```
func solveSingleElementBidirectionally(
    targetElementName string,
    recipe *scraper.Recipe,
    tierMap *scraper.Tier,
) scraper.TreeNode {

    if recipe == nil {
        globalNodeIdCounter++
        return scraper.TreeNode{Id: globalNodeIdCounter,
Name: "Error: Recipe data is nil."}
    }
    if scraper.IsBaseElement(targetElementName) {
        globalNodeIdCounter++
        return scraper.TreeNode{Id: globalNodeIdCounter,
Name: targetElementName, ImageSrc:
(*tierMap)[targetElementName].ImageSrc}
    }

    recipesForTarget, targetExists :=
(*recipe)[targetElementName]
    if !targetExists || len(recipesForTarget) == 0 {
        globalNodeIdCounter++
        return scraper.TreeNode{Id: globalNodeIdCounter,
Name: fmt.Sprintf("Element '%s' no recipes.", targetElementName)}
    }

    visitedFwd := make(map[string]bool)
    baseElementsList := getBaseElementsInternal(tierMap)
```

```

        if len(baseElementsList) == 0 {
            globalNodeIdCounter++
            return scraper.TreeNode{Id: globalNodeIdCounter,
Name: "Error: No base elements."}
        }

        currentFwdQueue := make([]string, 0,
len(baseElementsList))
        for _, el := range baseElementsList {
            if !visitedFwd[el] {
                visitedFwd[el] = true
                currentFwdQueue = append(currentFwdQueue, el)
            }
        }

        visitedBwd := make(map[string]bool)
        currentBwdQueue := []string{targetElementName}
        visitedBwd[targetElementName] = true

        collectedMeetingNodes := make(map[string]bool)
        orderedMeetingNodes := []string{}

        const maxSearchDepth = 20
        const maxReconstructionDepth = 15

        for depth := range maxSearchDepth {
            if len(orderedMeetingNodes) > 0 && depth > 2 {
                if len(orderedMeetingNodes) > 5 {
                    break
                }
            }

            nextFwdQueue := []string{}
            for _, nodeFromBase := range currentFwdQueue {
                if visitedBwd[nodeFromBase] &&
!collectedMeetingNodes[nodeFromBase] {
                    collectedMeetingNodes[nodeFromBase] =
true
                    orderedMeetingNodes =
append(orderedMeetingNodes, nodeFromBase)
                }
            }

            for product, combinations := range *recipe {
                // ... (Logic for forward BFS step) ...
            }
        }
    }
}

```

```

        for _, n := range nextFwdQueue {
            visitedFwd[n] = true
        }
        currentFwdQueue = nextFwdQueue

        nextBwdQueue := []string{}
        for _, nodeToMake := range currentBwdQueue {
            if visitedFwd[nodeToMake] &&
!collectedMeetingNodes[nodeToMake] {
                collectedMeetingNodes[nodeToMake] =
true
                orderedMeetingNodes =
append(orderedMeetingNodes, nodeToMake)
            }
            // ... (Logic for backward BFS step) ...
        }
        currentBwdQueue = nextBwdQueue

        if len(currentFwdQueue) == 0 && len(currentBwdQueue) ==
0 && depth > 1 {
            break
        }
    }

    if len(orderedMeetingNodes) == 0 {
        stopConditionStandard := func(name string) bool {
return scraper.IsBaseElement(name) }
        pathVisitedStandard := make(map[string]bool)
        return
reconstructSinglePathRecursive(targetElementName, recipe,
tierMap, stopConditionStandard, pathVisitedStandard, 0,
maxReconstructionDepth)
    }

    sort.Slice(orderedMeetingNodes, func(i, j int) bool {
        // ... (Sorting logic for meeting nodes based on
tier and name) ...
    })

    chosenMeetingNode := orderedMeetingNodes[0]

    stopConditionForSeg1 := func(name string) bool {
        return name == chosenMeetingNode ||
(scraper.IsBaseElement(name) && name != targetElementName)
    }
    pathVisitedSeg1 := make(map[string]bool)
    segmentStartToMeeting :=

```

```

reconstructSinglePathRecursive(targetElementName, recipe,
tierMap, stopConditionForSeg1, pathVisitedSeg1, 0,
maxReconstructionDepth)

    var segmentMeetingToBase scraper.TreeNode
    if scraper.IsBaseElement(chosenMeetingNode) {
        globalNodeIdCounter++
        segmentMeetingToBase = scraper.TreeNode{}
    } else {
        stopConditionForSeg2 := func(name string) bool {
            return scraper.IsBaseElement(name)
        }
        pathVisitedSeg2 := make(map[string]bool)
        segmentMeetingToBase =
reconstructSinglePathRecursive(chosenMeetingNode, recipe,
tierMap, stopConditionForSeg2, pathVisitedSeg2, 0,
maxReconstructionDepth)
    }

    if segmentStartToMeeting.Name == chosenMeetingNode {
        if len(segmentMeetingToBase.Children) > 0 {
            segmentStartToMeeting.Children =
segmentMeetingToBase.Children
        }
        return segmentStartToMeeting
    }

    nodeToAttach :=
findNodeInTreeBFSInternal(&segmentStartToMeeting,
chosenMeetingNode)

    if nodeToAttach != nil {
        if len(segmentMeetingToBase.Children) > 0 {
            nodeToAttach.Children =
segmentMeetingToBase.Children
        }
    }
    return segmentStartToMeeting
}

```

3. Fungsi reconstructSinglePathRecursive(currentNodeName string,
 recipeData *scraper.Recipe, tierMap *scraper.Tier, stopCondition func(name
 string) bool, pathVisited map[string]bool, currentDepth int,
 maxRecursiveDepth int) → scraperTreeNode

Fungsi rekursif yang digunakan untuk membangun (merekonstruksi) satu jalur pembuatan elemen dari currentNodeName hingga kondisi berhenti (stopCondition) terpenuhi atau kedalaman rekursi maksimum (maxRecursiveDepth) tercapai. Fungsi ini memilih resep pertama yang tersedia untuk currentNodeName dan secara rekursif memanggil dirinya sendiri untuk bahan-bahan dalam resep tersebut. pathVisited digunakan untuk mencegah siklus tak terbatas.

```
func reconstructSinglePathRecursive(  

    currentNodeName string,  

    recipeData *scraper.Recipe,  

    tierMap *scraper.Tier,  

    stopCondition func(name string) bool,  

    pathVisited map[string]bool,  

    currentDepth int,  

    maxRecursiveDepth int,  

) scraperTreeNode {  

    globalNodeIdCounter++  

    rootNode := scraperTreeNode{Id: globalNodeIdCounter,  

        Name: currentNodeName, ImageSrc:  

        (*tierMap)[currentNodeName].ImageSrc}  

    if currentDepth > maxRecursiveDepth {  

        return rootNode  

    }  

    if stopCondition(currentNodeName) {  

        return rootNode  

    }  

    if pathVisited[currentNodeName] {  

        return rootNode  

    }  

    pathVisited[currentNodeName] = true  

    defer delete(pathVisited, currentNodeName)  

    availableRecipes, recipesExist :=
```

```
(*recipeData)[currentNodeName]
    if !recipesExist || len(availableRecipes) == 0 {
        return rootNode
    }
    combination := availableRecipes[0]
    ingredient1Name := combination.First()
    ingredient2Name := combination.Second()

    if ingredient1Name == "" && ingredient2Name == "" {
        return rootNode
    }

    var childrenForPlusNode []scraper.TreeNode
    childPathVisited1 := make(map[string]bool)
    for k, v := range pathVisited {
        childPathVisited1[k] = v
    }

    if ingredient1Name != "" {
        child1Tree :=
        reconstructSinglePathRecursive(ingredient1Name, recipeData,
            tierMap, stopCondition, childPathVisited1, currentDepth+1,
            maxRecursiveDepth)
        childrenForPlusNode = append(childrenForPlusNode,
            child1Tree)
    }

    childPathVisited2 := make(map[string]bool)
    maps.Copy(childPathVisited2, pathVisited)
    if ingredient2Name != "" {
        child2Tree :=
        reconstructSinglePathRecursive(ingredient2Name, recipeData,
            tierMap, stopCondition, childPathVisited2, currentDepth+1,
            maxRecursiveDepth)
        childrenForPlusNode = append(childrenForPlusNode,
            child2Tree)
    }

    if len(childrenForPlusNode) > 0 {
        globalNodeIdCounter++
        plusNode := scraper.TreeNode{Id:
            globalNodeIdCounter, Name: "+", Children: childrenForPlusNode}
        rootNode.Children = append(rootNode.Children,
            plusNode)
    }
    return rootNode
}
```

4. Fungsi `getBaseElementsInternal(tierMap *scraper.Tier) → []string`

Fungsi pembantu untuk mendapatkan daftar elemen dasar. Elemen dasar didefinisikan sebagai elemen dengan Tier = 0 dalam tierMap. Jika tierMap kosong atau tidak valid, fungsi ini mengembalikan daftar elemen dasar yang sudah diketahui secara *hardcoded* (Air, Earth, Fire, Water, Time). Hasilnya diurutkan secara alfabetis.

```
func getBaseElementsInternal(tierMap *scraper.Tier) []string {
    var baseElements []string
    knownBaseElements := []string{"Air", "Earth", "Fire",
    "Water", "Time"}

    if tierMap != nil && len(*tierMap) > 0 {
        tempBaseElementsMap := make(map[string]bool)
        for el, info := range *tierMap {
            tierVal := info.Tier
            if tierVal == 0 {
                tempBaseElementsMap[el] = true
            }
        }

        if len(tempBaseElementsMap) > 0 {
            for el := range tempBaseElementsMap {
                baseElements = append(baseElements, el)
            }
            sort.Strings(baseElements)
            return baseElements
        }
    }
    sort.Strings(knownBaseElements)
    return knownBaseElements
}
```

5. Fungsi `findNodeInTreeBFSInternal(root *scraper.TreeNode, name string) → *scraper.TreeNode`

Fungsi pembantu yang melakukan pencarian BFS (*Breadth First Search*) pada pohon `scraper.TreeNode` yang sudah ada (root) untuk menemukan *node* dengan nama (name) tertentu. Fungsi ini digunakan untuk menemukan *meeting node* dalam pohon yang sudah direkonstruksi sebagian (`segmentStartToMeeting`) agar bagian kedua dari jalur (`segmentMeetingToBase`) dapat disambungkan.

```

func findNodeInTreeBFSInternal(root *scraper.TreeNode, name
string) *scraper.TreeNode {
    if root == nil {
        return nil
    }
    q := []*scraper.TreeNode{root}

    for len(q) > 0 {
        curr := q[0]
        q = q[1:]

        if curr.Name == name {
            return curr
        }

        for i := range curr.Children {
            childNode := &curr.Children[i]
            if childNode.Name == "+" {
                for j := range childNode.Children {
                    q = append(q,
&childNode.Children[j])
                }
            }
        }
    }
    return nil
}

```

6. Fungsi countNodesInTree(node *scraper.TreeNode) → int

Rekursif sederhana untuk menghitung jumlah node elemen dalam pohon scraper.TreeNode yang diberikan.

```

func countNodesInTree(node *scraper.TreeNode) int {
    if node == nil {
        return 0
    }
    count := 0
    if node.Name != "+" {
        count = 1
    }

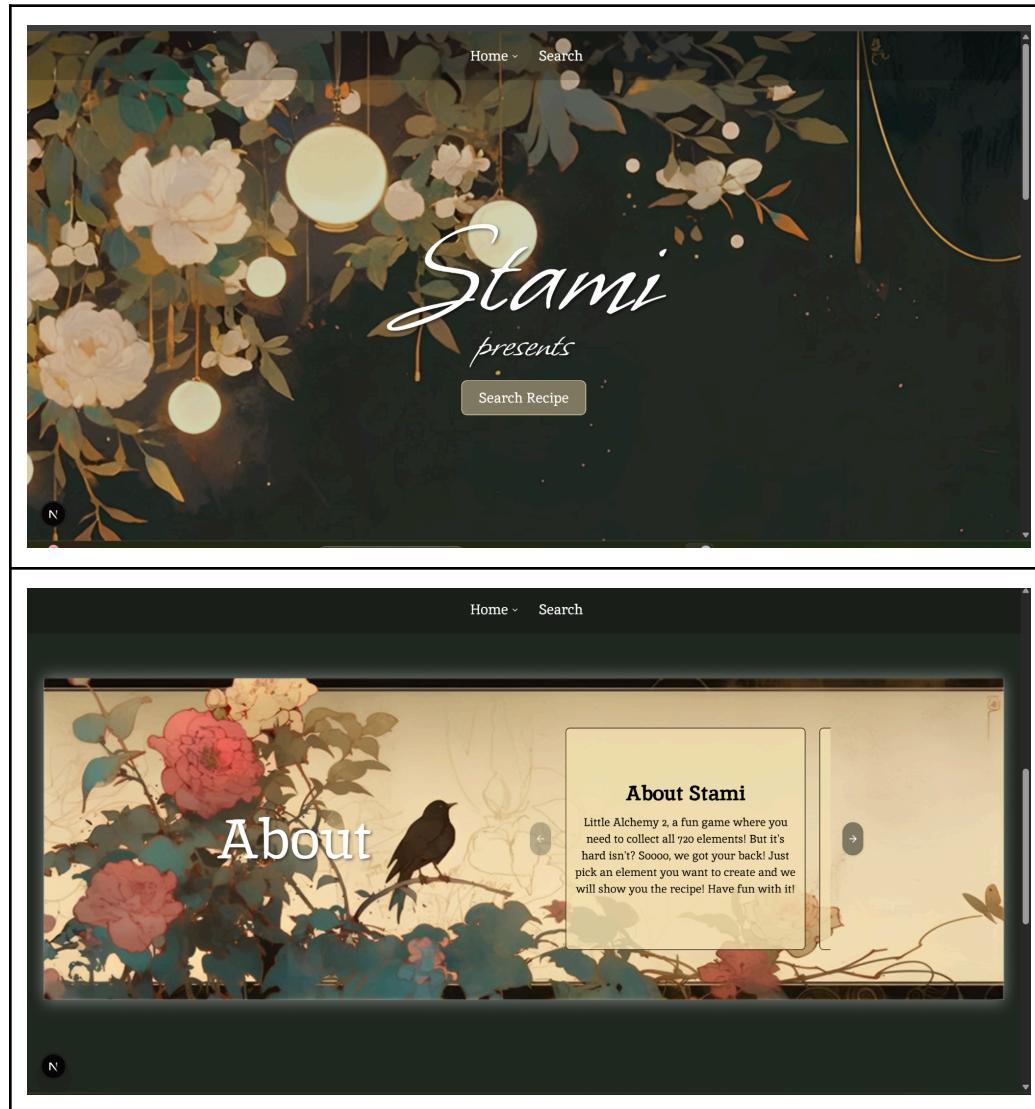
    for i := range node.Children {
        count += countNodesInTree(&node.Children[i])
    }
    return count
}

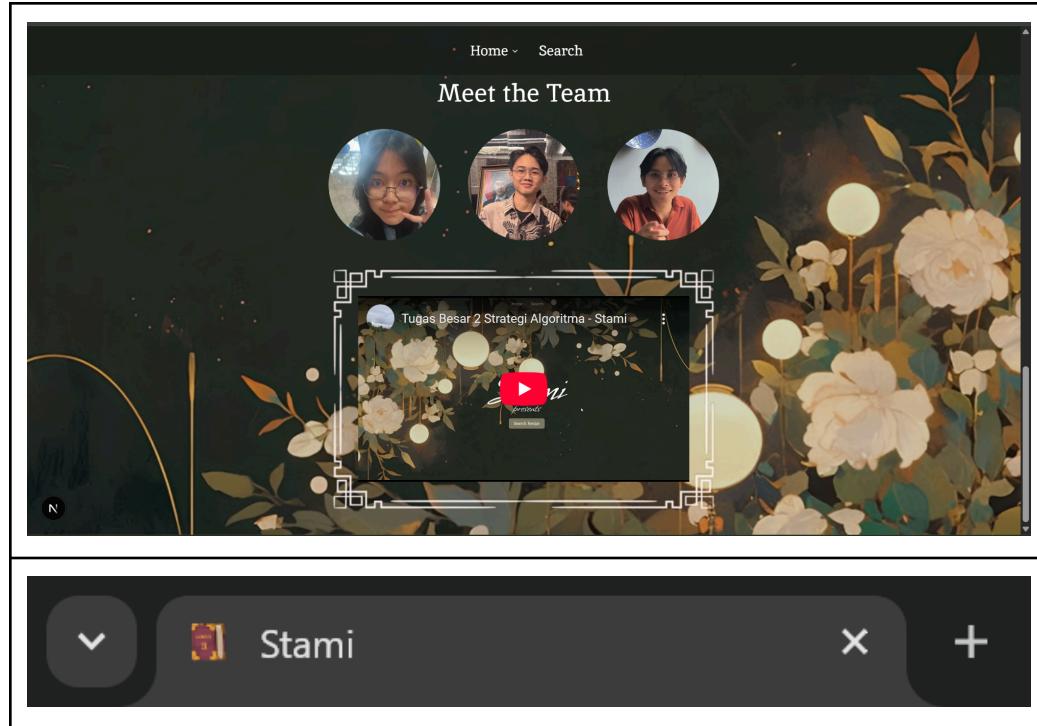
```

4.4 Tata Cara Penggunaan Program

4.4.1 Laman Utama

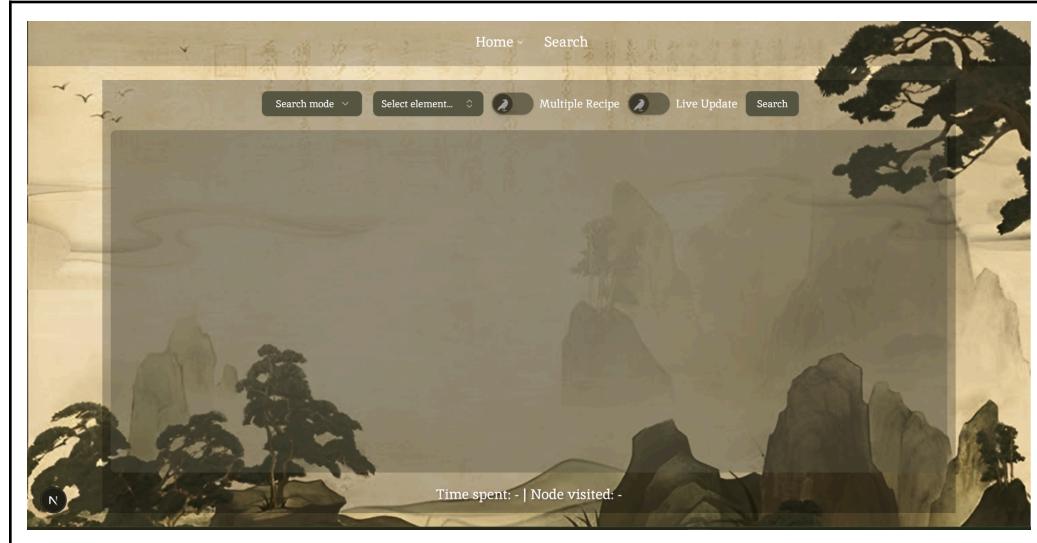
Laman utama berisi tiga bagian, *home* yang akan mengarahkan pada laman pencarian, *about* yang berisi informasi mengenai program, cara menggunakan, dan penjelasan singkat mengenai algoritma pencarian, dan *team* yang menunjukkan anggota kelompok dan hasil video penjelasan kreatif yang dibuat oleh kelompok.

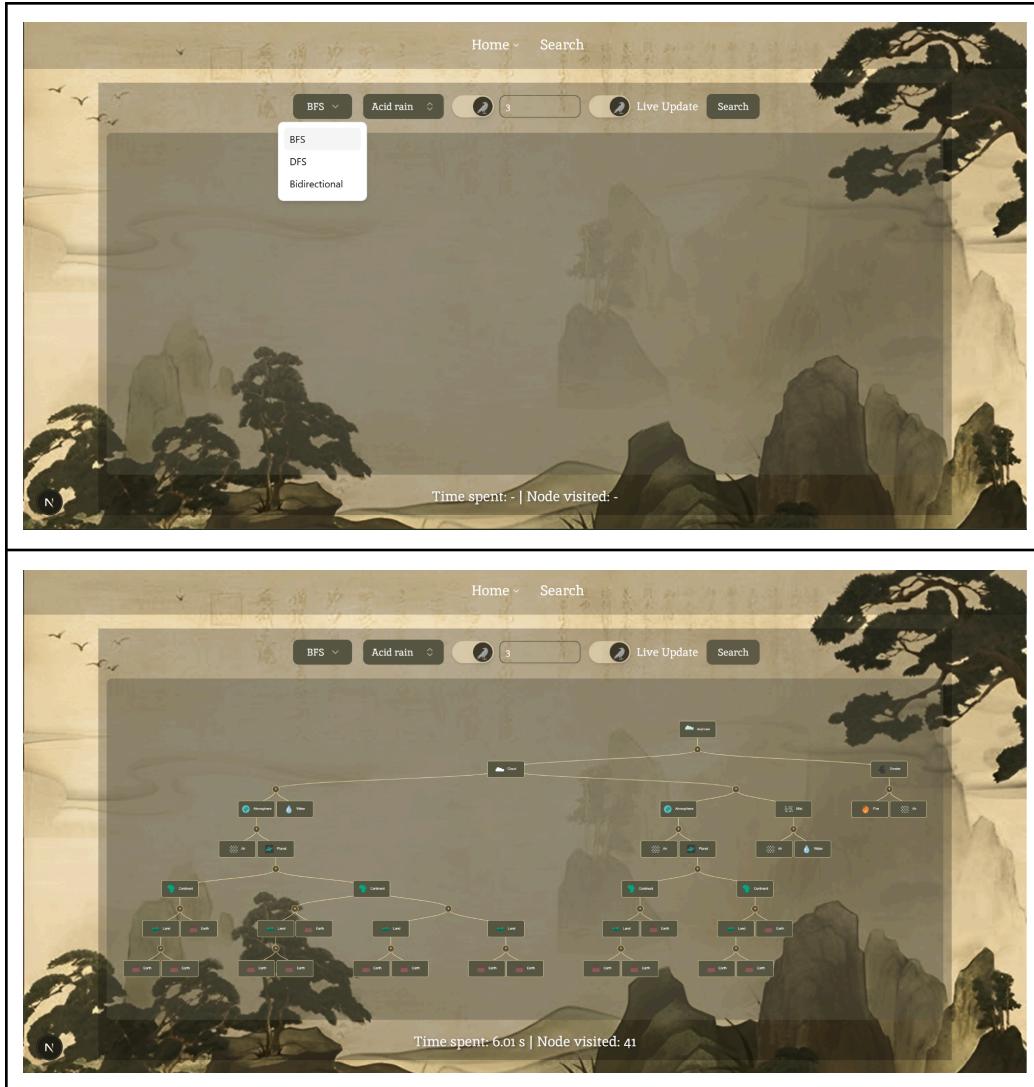




4.4.2 Laman Pencarian

Laman ini merupakan inti dari program pencarian. Pengguna dapat memilih algoritma pencarian yang diinginkan, elemen yang ingin dicari resepnya, memasukkan banyak resep yang diinginkan, dan memilih untuk melihat *live update* pencarian atau tidak.





4.5 Hasil Pengujian

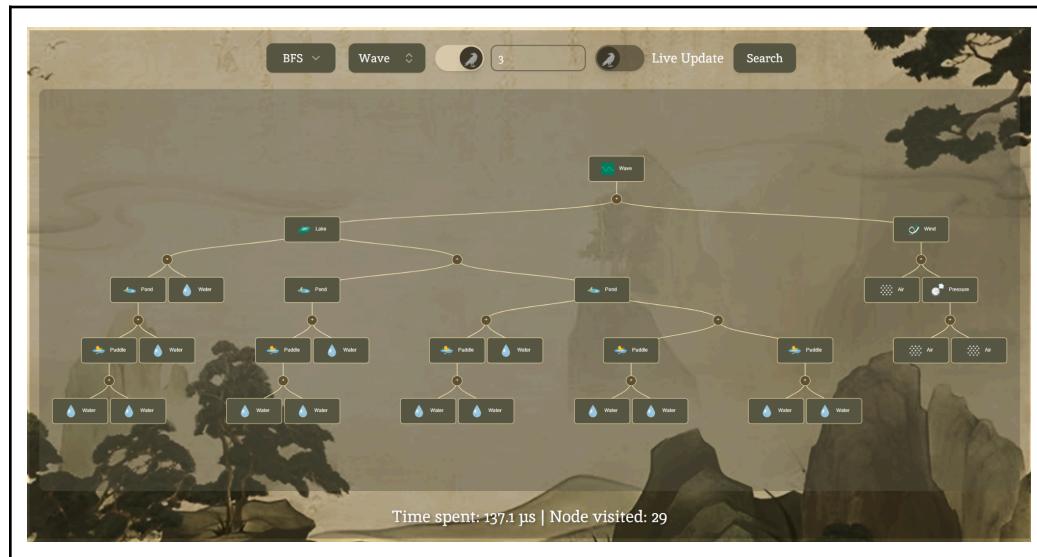
4.5.1 Test Case 1

Pengujian BFS satu resep



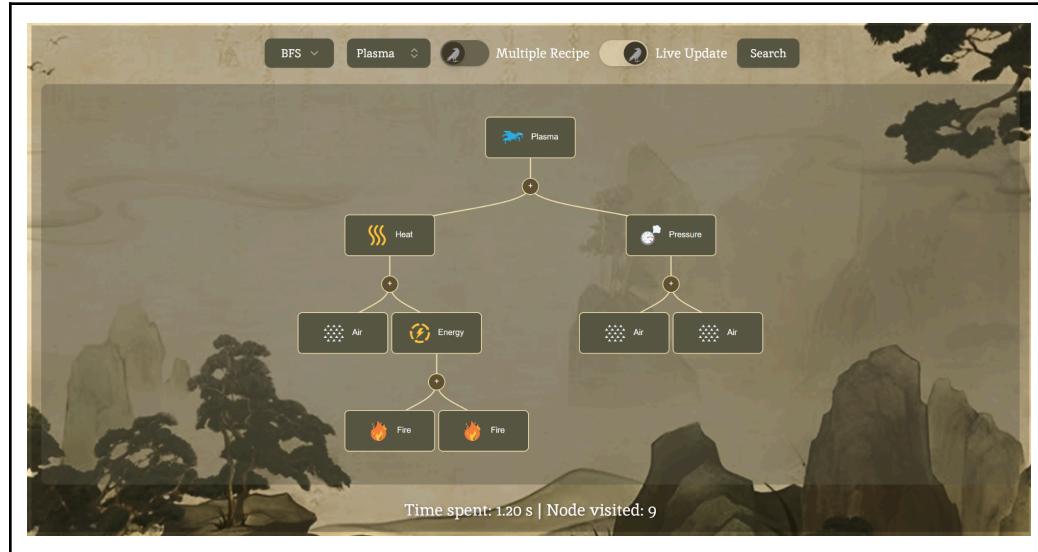
4.5.2 Test Case 2

Pengujian BFS banyak resep



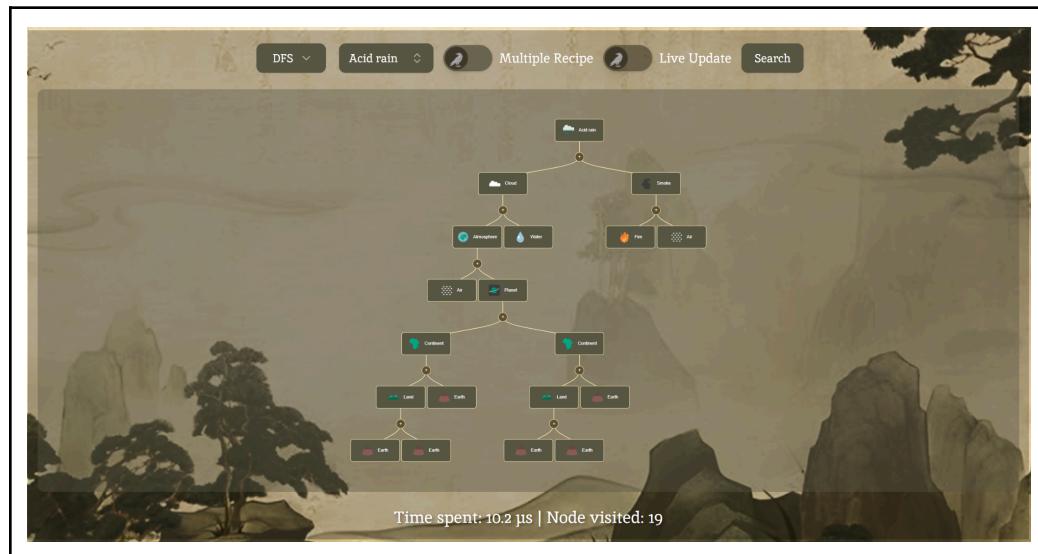
4.5.3 Test Case 3

Pengujian BFS *live update*



4.5.4 Test Case 4

Pengujian DFS satu resep



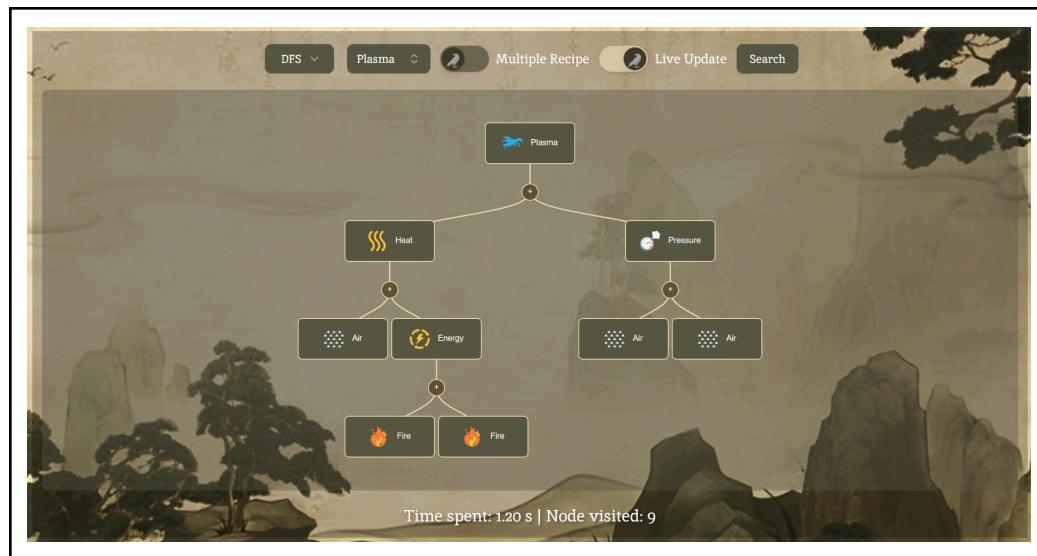
4.5.5 Test Case 5

Pengujian DFS banyak resep



4.5.6 Test Case 6

Pengujian DFS *live update*



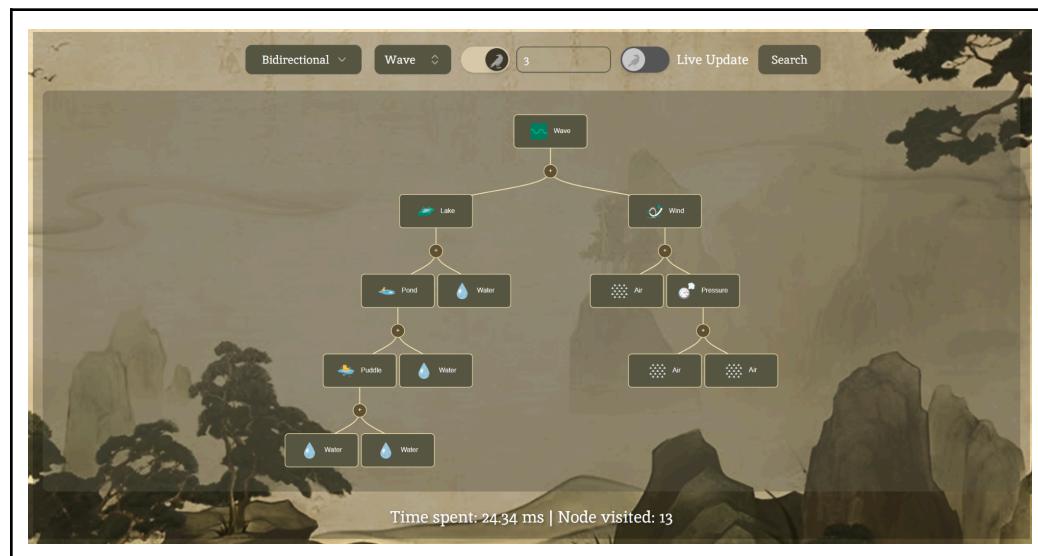
4.5.7 Test Case 7

Pengujian *Bidirectional* satu resep



4.5.8 Test Case 8

Pengujian *Bidirectional* banyak resep



4.6 Analisis Hasil Pengujian

Terdapat beberapa pengujian yang dilakukan terhadap elemen yang ingin dicari resepnya. Dari *live update*, proses pencarian dan implementasi algoritma dapat terlihat dengan jelas. BFS yang menggunakan metode iteratif, mengecek elemen pembuat dari suatu elemen. DFS yang menggunakan metode rekursif, mengecek salah satu elemen

pembuat lebih mendalam sampai ke elemen dasar. *Bidirectional* yang menggunakan metode iteratif, melakukan pencarian dari dua arah secara bersamaan, untuk yang kami kerjakan, pencarian maju dan mundur menggunakan BFS, satu elemen target mundur ke elemen dasar, dan satu lagi dari elemen-elemen dasar menuju elemen target, hingga kedua pencarian tersebut bertemu di satu atau beberapa elemen perantara untuk membentuk jalur solusi.

Dapat dilihat bahwa algoritma DFS cenderung lebih cepat dibandingkan dengan algoritma pencarian lainnya. Hal ini terjadi karena DFS yang merupakan pencarian mendalam sehingga *goal node* (dalam kasus ini elemen dasar) dapat ditemui lebih dahulu dibandingkan BFS yang mengunjungi atau mencari *goal node* per *level*. Mengenai jumlah *node* yang dikunjungi, setiap algoritma pencarian cenderung mempunyai jumlah yang sama. Selain itu, dengan fitur *live update*, waktu pencarian menjadi lebih lama dikarenakan perhitungan tidak hanya pada pencarian namun mengikuti kecepatan *live update* berjalan.

BAB V

PENUTUP

5.1 Kesimpulan

Melalui Tugas Besar II Strategi Algoritma, kami belajar cara pengembangan suatu *website* dengan menggunakan *repository* yang berbeda untuk pengembangan *frontend* dan *backend*. Kami belajar lebih lagi mengenai bahasa Go dan JavaScript serta *framework-framework* yang kami gunakan, seperti Gin dan NextJS. Selain itu, kami juga mempelajari cara menulis dan mengimplementasikan algoritma BFS, DFS, dan *Bidirectional* secara langsung. Hasil pengujian yang dilakukan pun membuat kami mengetahui bahwa algoritma DFS merupakan algoritma yang tercepat untuk mendapatkan hasil dengan pengecualian untuk beberapa kasus.

5.2 Saran

Berikut adalah saran dan refleksi dari kelompok untuk kelompok di masa depan.

1. Berkomunikasi dengan lebih aktif untuk proses implementasi.
2. Mengoptimalkan dan membuat pembagian tugas yang lebih jelas serta meningkatkan kerjasama tim.
3. Peningkatan dalam penulisan komentar dan dokumentasi untuk mempermudah anggota kelompok lain untuk memahami yang juga berguna untuk *maintenance* program.

5.3 Refleksi

Pengerjaan Tugas Besar 2 Strategi Algoritma memberikan pengetahuan dan pengalaman mengenai algoritma pencarian graf, seperti BFS, DFS, dan *Bidirectional*. Kelompok juga menyadari bahwa komunikasi merupakan hal yang penting dalam pengerjaan tugas ini.

LAMPIRAN

- Tautan repository Github
 - Frontend
https://github.com/BerthaSoliany/Tubes2_FE_Stami
 - Backend
https://github.com/albertchriss/Tubes2_BE_stami
- Video
https://youtu.be/tEMz_p-40Pk
- Tabel

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

Dicoding Intern. (2020). *Apa Itu JavaScript? Fungsi dan Contohnya*. Diakses pada 11 Mei 2025 melalui <https://www.dicoding.com/blog/apa-itu-javascript-fungsi-dan-contohnya/>.

Rinaldi, M. (2025). *Breadth/Depth First Search (BFS/DFS) Bag1*. Diakses pada 11 Mei 2025 melalui

[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).

GO. *Go Documentation*. Diakses pada 10 Mei 2025 melalui <https://go.dev/doc/>.

Gin Web Framework. *Gin Documentation*. Diakses pada 10 Mei 2025 melalui <https://gin-gonic.com/docs/>.

Docker. *Docker: Accelerated Container Application Development*. Diakses pada 10 Mei 2025 melalui <https://www.docker.com/#build>.

Shadcn/ui. *Introduction*. Diakses pada 10 Mei 2025 melalui <https://ui.shadcn.com/docs>.

JagoanHosting_. (2022). *Web Development: Pengertian, Jenis, dan Proses Kerjanya*. Diakses pada 10 Mei 2025 melalui <https://www.jagoanhosting.com/blog/web-development/>.

GeeksforGeeks. (2024). *Bidirectional Search*. Diakses pada 13 Mei 2025 melalui <https://www.geeksforgeeks.org/bidirectional-search/>.

RevoUpedia. *Apa itu deployment?*. Diakses pada 13 Mei 2025 melalui <https://www.revou.co/kosakata/deployment>.

NV. (2023). *Sebelum Membuat Situs Web, Ketahui Fase Pengembangan Web yang Harus Dilalui*. Diakses pada 13 Mei 2025 melalui <https://totalit.co.id/blog/sebelum-membuat-situs-web-ketahui-fase-pengembangan-web-yang-harus-dilalui>.