Tugas Besar 2 IF2211 Strategi Algoritma

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

Oleh:

Kelompok 3 – TrioKwekKwek

Azadi Azhrah (12823024)

Sebastian Enrico Nathanael (13523134)

Bryan P. Hutagalung (18222130)

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika – Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

2025

Daftar Isi

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.

Komponen-komponen dari permainan ini antara lain:

1. **Elemen dasar**

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

2. **Elemen turunan**

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

3. **Combine Mechanism**

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

Bab II

Landasan Teori

2.1 Algoritma Graph Search

Dalam permainan **Little Alchemy 2**, graf berfungsi sebagai struktur yang menghubungkan berbagai elemen yang bisa digabungkan untuk menghasilkan elemen baru. Elemen dasar seperti **Air, Fire, Earth**, dan **Water** merupakan node dasar dalam graf ini, sementara kombinasi antara elemen-elemen tersebut membentuk edge yang menghubungkan node satu dengan node lainnya. Algoritma Graph Search digunakan untuk menemukan kombinasi yang benar dan menghasilkan elemen-elemen baru berdasarkan resep yang ada dalam permainan. Dua algoritma pencarian yang paling relevan untuk menyelesaikan masalah ini adalah **Breadth-First Search (BFS)** dan **Depth-First Search (DFS)**, yang akan dijelaskan lebih rinci berikut ini.

2.1.1 Breadth-First Search

1. Definisi dan cara kerja algoritma BFS

Breadth-First Search (BFS) adalah algoritma pencarian yang digunakan untuk mengeksplorasi graf dengan cara meluas secara bertahap, mulai dari node awal dan melanjutkan ke node yang berada pada kedalaman yang lebih tinggi. Dalam konteks Little Alchemy 2, BFS dimulai dengan elemen dasar seperti Air, Fire, Earth, dan Water, kemudian secara berurutan mencoba semua kombinasi yang mungkin untuk menghasilkan elemen baru. Pencarian dilakukan dengan menggunakan queue untuk menyimpan node yang akan dikunjungi.

Cara kerja BFS secara umum

- Inisialisasi queue dengan node awal (elemen dasar).
- Ambil node dari depan queue dan kunjungi.
- Tambahkan tetangga node tersebut yang belum dikunjungi ke dalam queue.
- Ulangi langkah 2 dan 3 hingga queue kosong atau node tujuan ditemukan.

Dalam implementasinya di **Little Alchemy 2**, BFS memungkinkan eksplorasi elemen-elemen dasar yang mengarah pada elemen-elemen lebih kompleks hingga mencapai elemen target.

2. **Kompleksitas Waktu dan Ruang BFS**
    - **Kompleksitas Waktu**

      O(V + E), di mana V adalah jumlah node (elemen) dan E adalah jumlah edge (kombinasi). Dalam kasus terburuk, BFS akan mengunjungi setiap node dan edge.
    - **Kompleksitas Ruang**

      O(V), yang mencerminkan jumlah elemen yang harus disimpan dalam memori untuk melakukan pencarian.
3. **Kelebihan dan Kekurangan BFS dalam Pencarian Recipe**
    - **Kelebihan**
        1) Menjamin menemukan jalur terpendek, yaitu kombinasi elemen dengan langkah minimum.
        2) Efektif dalam menemukan solusi optimal untuk elemen yang membutuhkan banyak kombinasi dengan tier rendah hingga menengah.
        3) Memungkinkan identifikasi berbagai jalur alternatif pada tingkat kedalaman yang sama.
    - **Kekurangan**
        1) Membutuhkan banyak memori untuk menyimpan node pada tingkat yang sama.
        2) Kurang efisien untuk elemen kompleks dengan banyak kombinasi, seperti elemen dengan tier tinggi.
        3) Dapat mengeksplorasi jalur yang tidak relevan jika graf memiliki banyak cabang.
4. **Implementasi BFS pada Little Alchemy 2**

    BFS dimulai dengan menambahkan elemen dasar seperti **Air, Fire, Earth,** dan **Water** ke dalam queue. Setiap elemen kemudian akan dikombinasikan dengan elemen lain yang sudah ditemukan. Jika kombinasi baru berhasil, elemen tersebut ditambahkan ke dalam

queue untuk dieksplorasi lebih lanjut. Proses ini diulang hingga elemen target ditemukan.

Beberapa optimasi yang diterapkan:

- **Hierarki Tier**
  Menghindari eksplorasi elemen yang lebih rendah tier-nya dibandingkan target.
- **Pencarian Berdasarkan Hierarki**
  Mengutamakan elemen dengan tier lebih tinggi.
- **Backtracking**
  Menyimpan informasi tentang parent node untuk merekonstruksi jalur pencarian.

2.1.2 Depth-First Search

1. **Definisi dan cara kerja algoritma DFS**

   Depth-First Search (DFS) adalah algoritma yang mengeksplorasi graf dengan mendalam, berusaha mencapai kedalaman maksimum dalam satu cabang graf sebelum mundur dan melanjutkan ke cabang lainnya. DFS menggunakan stack atau mekanisme rekursif untuk melacak node yang dikunjungi.

   Cara kerja DFS secara umum:

   - Mulai dari node awal dan tandai sebagai dikunjungi.
   - Jelajahi tetangga pertama yang belum dikunjungi.
   - Ulangi proses untuk tetangga tersebut hingga kedalaman graf tercapai.
   - Jika mencapai node tanpa tetangga yang belum dikunjungi, lakukan **backtrack** ke node sebelumnya.
   - Proses berakhir ketika semua node telah dijelajahi atau node tujuan ditemukan.

   Di **Little Alchemy 2**, DFS berusaha menemukan kombinasi elemen yang lebih mendalam, yang dapat menghasilkan rantai elemen yang lebih kompleks sebelum mencoba alternatif lainnya.

2. **Kompleksitas waktu dan ruang DFS**

- **Kompleksitas Waktu**

  O(V + E), serupa dengan BFS, DFS mengunjungi setiap node dan edge.

- **Kompleksitas Ruang**

  O(h), di mana h adalah kedalaman maksimum dari pencarian, yang berhubungan langsung dengan panjang rantai elemen yang dieksplorasi.

3. **Kelebihan dan Kekurangan DFS dalam Pencarian Recipe**
   - **Kelebihan**
     1) Memiliki penggunaan memori yang lebih efisien dibandingkan BFS.
     2) Cocok untuk menemukan solusi dengan cepat, tanpa harus mengeksplorasi seluruh ruang solusi.
     3) Lebih efektif untuk kombinasi elemen yang melibatkan rantai panjang.
   - **Kekurangan**
     1) Tidak menjamin jalur terpendek, karena pencarian bisa saja berakhir pada jalur yang tidak optimal.
     2) Rentan terhadap **stack overflow** jika graf terlalu dalam.
     3) Cenderung terjebak dalam cabang panjang tanpa menemukan solusi.
     4) Membutuhkan mekanisme untuk mencegah siklus dalam graf.
4. **Implementasi DFS pada problem space Little Alchemy 2**

   Pada implementasi DFS, pencarian dilakukan secara rekursif. Setiap elemen dasar dicoba dengan berbagai kombinasi untuk membentuk elemen yang lebih kompleks. DFS lebih menekankan eksplorasi mendalam pada setiap cabang pencarian, dengan beberapa batasan kedalaman untuk mencegah pencarian yang terlalu dalam.

   Strategi implementasi DFS:

- **Batasan Kedalaman Maksimum**

  Biasanya disesuaikan dengan perkiraan 2× dari tier elemen target.
- **Pencegahan Siklus**

  Menggunakan struktur data seperti hash map untuk melacak elemen yang telah dikunjungi.
- **Prioritas Komposisi**

  Mengutamakan elemen dengan tier lebih tinggi untuk mengoptimalkan pencarian.

2.1.3 Bidirectional Search

1. Definisi dan Cara Kerja Bidirectional Search

Bidirectional Search adalah strategi pencarian yang melakukan dua pencarian secara simultan: satu dimulai dari node awal dan satu lagi dari node tujuan. Pencarian berakhir ketika kedua pencarian bertemu di tengah, sehingga jalur dari node awal ke node tujuan dapat dibangun dengan lebih efisien.

Cara kerja Bidirectional Search:

- Mulai dengan dua pencarian: satu dari node awal dan satu dari node tujuan.
- Secara bergantian lakukan pencarian dari kedua arah.
- Cek apakah ada node yang telah dikunjungi oleh kedua pencarian.
- Jika menemukan node yang sama, rekonstruksi jalur dengan menggabungkan jalur dari kedua pencarian.

Di **Little Alchemy 2**, pencarian dapat dilakukan baik dari elemen dasar menuju elemen target (forward) maupun sebaliknya, dari elemen target mencari elemen-elemen pembentuknya (backward).

2. Kompleksitas Waktu dan Ruang

- **Kompleksitas Waktu**

  O(b^(d/2)), di mana b adalah faktor percabangan rata-rata dan d adalah panjang jalur terpendek. Dengan pencarian dari dua arah, pencarian menjadi jauh lebih efisien dibandingkan pencarian satu arah (O(b^d)).

- **Kompleksitas Ruang**

  O(b^(d/2)), yang mencerminkan kebutuhan untuk menyimpan frontier dari kedua arah pencarian.

3. **Kelebihan dan Kekurangan Bidirectional Search**
   - **Kelebihan**
     1) Secara signifikan mengurangi ruang pencarian, meningkatkan efisiensi.
     2) Dapat menemukan jalur optimal, terutama jika menggunakan BFS pada kedua arah pencarian.
     3) Lebih efisien pada graf dengan percabangan tinggi seperti yang ada dalam **Little Alchemy 2**.
   - **Kekurangan**
     1) Implementasi lebih kompleks, terutama dalam menemukan titik temu antara dua pencarian.
     2) Memerlukan kemampuan untuk melakukan pencarian mundur, yang membutuhkan mapping elemen hasil ke elemen pembentuknya.
     3) Membutuhkan lebih banyak memori dibandingkan DFS sederhana.
4. **Implementasi Bidirectional Search pada Little Alchemy 2**

   Bidirectional Search diimplementasikan dengan menggunakan dua queue: satu untuk pencarian maju (forward) dari elemen dasar dan satu lagi untuk pencarian mundur (backward) dari elemen target. Pencarian dilanjutkan hingga kedua arah bertemu di titik yang sama, di mana jalur pencarian dapat direkonstruksi.

2.2 Aplikasi Website

Aplikasi **Little Alchemy 2 Recipe Finder** dibangun dengan menggunakan berbagai teknologi web yang memungkinkan aplikasi berjalan dengan efisien dan mudah diakses. Dalam pengembangannya, kami memilih **Next.js** untuk frontend dan **Go (Golang)** untuk backend, serta memanfaatkan **Docker** untuk kemudahan deployment. Berikut adalah penjelasan lebih rinci tentang pemilihan teknologi dan penerapannya.

2.2.1 Frontend

React adalah sebuah pustaka JavaScript yang digunakan untuk membangun antarmuka pengguna berbasis komponen. React memungkinkan pembuatan UI yang interaktif dan deklaratif, yang berarti pengembang hanya perlu mendeskripsikan bagaimana tampilan harus berubah berdasarkan state aplikasi. Dalam konteks **Little Alchemy 2 Recipe Finder**, React memudahkan pengembangan komponen yang dapat digunakan kembali, seperti form pencarian, visualisasi tree, dan pemilihan algoritma pencarian. Sifatnya yang berbasis komponen memungkinkan pemeliharaan kode yang lebih mudah dan kolaborasi antar pengembang.

Next.js adalah framework yang dibangun di atas React dan memperluas kemampuannya dengan menyediakan fitur-fitur tambahan seperti **server-side rendering** (SSR), **static site generation** (SSG), **routing berbasis file**, dan **API routes**. Next.js sangat cocok digunakan dalam tugas besar ini karena beberapa alasan:

- **Server-side Rendering (SSR)**
  Memungkinkan aplikasi untuk menghasilkan halaman secara dinamis di sisi server sebelum dikirim ke browser. Ini meningkatkan performa aplikasi, terutama dalam hal kecepatan muat halaman, serta membantu optimasi SEO karena konten dapat langsung dilihat oleh mesin pencari.
- **API Routes**
  Next.js mendukung pembuatan API routes, yang memfasilitasi komunikasi antara frontend dan backend tanpa perlu mengatur CORS (Cross-Origin Resource Sharing). Ini sangat berguna untuk integrasi dengan backend yang dibangun menggunakan Go.
- **File-based Routing**
  Dengan Next.js, routing dilakukan berdasarkan struktur file dalam tugas besar, yang menyederhanakan navigasi dan pengorganisasian aplikasi.
- **Hot Reloading**
  Fitur ini memungkinkan pengembang untuk melihat perubahan

yang dilakukan pada kode secara langsung tanpa perlu memuat ulang seluruh aplikasi. Ini meningkatkan kecepatan dalam fase pengembangan.

1. **Komponen-komponen Utama dalam Aplikasi**

   Aplikasi **Recipe Finder** menggunakan beberapa komponen utama yang bekerja bersama untuk menyediakan fungsionalitas dan tampilan yang diperlukan:

   - **SearchForm**

     Komponen ini memungkinkan pengguna untuk memasukkan elemen target yang ingin dicari, serta parameter pencarian seperti algoritma pencarian dan mode pencarian.
   - **AlgorithmSelector**

     Memberikan opsi untuk memilih algoritma pencarian (BFS, DFS, atau Bidirectional).
   - **ModeSelector**

     Memungkinkan pengguna untuk memilih apakah mereka ingin mencari satu recipe atau banyak recipe sekaligus.
   - **RecipeTree**

     Komponen ini berfungsi untuk menampilkan visualisasi dari tree pencarian, menggambarkan bagaimana elemen-elemen dikombinasikan untuk menghasilkan elemen target.
   - **RecipeSteps**

     Menampilkan langkah-langkah kombinasi elemen untuk mencapai elemen target.
   - **SearchStats**

     Menyediakan informasi mengenai waktu pencarian dan jumlah node yang dikunjungi.

2. **Reactflow untuk Visualisasi Tree**

   Reactflow adalah pustaka React yang digunakan untuk membuat visualisasi berbasis node. Dalam tugas besar ini, Reactflow digunakan untuk memvisualisasikan **recipe tree**, yang menunjukkan bagaimana kombinasi elemen-elemen dapat menghasilkan elemen target. Fitur-fitur Reactflow yang digunakan mencakup:

- **Custom Node Types**

  Menampilkan berbagai jenis node dengan tampilan yang berbeda sesuai dengan peranannya, seperti elemen dasar, elemen turunan, dan hasil kombinasi.
- **Edge Styling**

  Menyesuaikan gaya garis koneksi antar node dengan animasi untuk menggambarkan proses pembuatan elemen.
- **Interactive Controls**

  Pengguna dapat melakukan zoom dan pan untuk menjelajahi tree, serta mengatur layout untuk memudahkan navigasi.
- **MiniMap**

  Memberikan gambaran umum tentang struktur tree, yang mempermudah pengguna saat tree tersebut sangat besar dan kompleks.
- **Layout Engine**

  Memungkinkan penataan otomatis node untuk memastikan hierarki yang jelas, dengan elemen target di bagian atas dan elemen dasar di bagian bawah.

2.2.2 Backend

Go (atau Golang) adalah bahasa pemrograman yang dikembangkan oleh Google yang terkenal karena efisiensinya dalam menangani operasi intensif serta performa tinggi. Go sangat cocok untuk digunakan pada bagian backend aplikasi ini karena karakteristik-karakteristik berikut:

- **Static Typing**

  Dengan penggunaan tipe data statis, Go mengurangi potensi kesalahan pada waktu runtime, memberikan kode yang lebih stabil dan mudah untuk dikelola.
- **Garbage Collection**

  Manajemen memori otomatis yang efisien memungkinkan pengelolaan memori yang lebih baik, tanpa perlu memikirkan pengelolaan memori manual.

- **Kompilasi Cepat**

  Go memiliki waktu kompilasi yang cepat, yang mempercepat siklus pengembangan dan memungkinkan pengujian serta iterasi lebih cepat.
- **Konkurensi Bawaan**

  Salah satu fitur utama Go adalah kemampuannya untuk menangani konkurensi menggunakan **goroutines** dan **channels**. Fitur ini sangat penting dalam optimasi pencarian multiple recipe yang dilakukan secara paralel.
- **Standard Library yang Kaya**

  Go dilengkapi dengan pustaka standar yang kuat, termasuk **HTTP server**, **JSON parsing**, dan fitur I/O, yang sangat berguna dalam membangun API dan menangani data.

1. **Implementasi REST API dengan Go**

   Backend aplikasi **Recipe Finder** menggunakan Go untuk menyediakan REST API yang memungkinkan frontend berkomunikasi dengan backend. API ini mengelola request pencarian elemen dan mengembalikan hasil dalam format JSON. Beberapa komponen dalam implementasi REST API termasuk:

   - **HTTP Server**

     Menggunakan package **net/http** dari Go untuk menangani permintaan HTTP.
   - **Route Handlers**

     Fungsi yang menangani berbagai request pencarian recipe, seperti mencari recipe berdasarkan algoritma yang dipilih.
   - **JSON Serialization**

     Mengonversi data dalam bentuk struktur Go menjadi format JSON untuk dikirimkan sebagai respons.
   - **Parameter Validation**

     Memastikan bahwa input dari pengguna (seperti elemen target dan algoritma) valid sebelum diproses.
   - **Error Handling**

     Menangani kesalahan dengan mengembalikan status code

HTTP yang sesuai untuk memberikan feedback yang jelas kepada pengguna.

2. **Multithreading di Go untuk Optimasi Pencarian**

   Go memungkinkan implementasi multithreading yang efisien dengan menggunakan **goroutines** untuk menjalankan pencarian variasi recipe secara paralel. Dalam implementasi ini:

   - **Goroutines**

     Digunakan untuk menjalankan pencarian berbagai variasi recipe secara terpisah, meningkatkan efisiensi.
   - **WaitGroup**

     Menyinkronkan penyelesaian goroutines yang aktif untuk memastikan semua pencarian selesai sebelum menggabungkan hasilnya.
   - **Mutex**

     Digunakan untuk menghindari kondisi balapan saat mengakses shared resources, seperti hasil pencarian.
   - **Channels**

     Membatasi jumlah goroutines aktif untuk menghindari pemborosan sumber daya dan menjaga stabilitas sistem.
   - **Context**

     Menangani pembatalan dan timeout pencarian yang terlalu lama.

2.2.3 Docker dan Containerization

Docker adalah platform containerization yang memungkinkan aplikasi untuk dijalankan dalam lingkungan terisolasi, yang disebut **container**. Container memastikan bahwa aplikasi berjalan konsisten di berbagai lingkungan, mengeliminasi masalah terkait perbedaan konfigurasi antara development, staging, dan production. Konsep utama Docker meliputii.

- **Container**

  Lingkungan terisolasi untuk menjalankan aplikasi beserta semua dependensinya.

- **Image**

  Template yang digunakan untuk membuat container, berisi semua dependencies yang diperlukan.
- **Dockerfile**

  Skrip yang berisi instruksi untuk membangun image.
- **Registry**

  Repositori untuk menyimpan dan mendistribusikan image.
- **Compose**

  Alat yang memungkinkan pengelolaan aplikasi multi-container.

1. **Struktur Docker untuk Frontend dan Backend**

   Proyek Recipe Finder menggunakan pendekatan multi-container dengan Docker Compose untuk menjalankan frontend dan backend secara terpisah namun tetap terhubung:

   - **Dockerfile untuk Frontend**

     Menggunakan image dasar **node:20-alpine**, dioptimalkan dengan multistage build untuk mengurangi ukuran image. Proses build mencakup instalasi dependencies dan pembuatan aplikasi Next.js.
   - **Dockerfile untuk Backend**

     Menggunakan image **golang:1.22-alpine** untuk proses build dan **alpine:latest** untuk image akhir yang minimalis. Aplikasi Go dikompilasi dan file data elemen dipindahkan ke dalam container.
   - **Docker Compose**

     Digunakan untuk mendefinisikan service untuk frontend dan backend, mengatur dependensi antar service, serta mengelola jaringan dan volume untuk memastikan komunikasi yang efisien antar container.
2. **Manfaat Penggunaan Docker dalam Deployment**

   Penggunaan Docker dalam tugas besar ini memberikan beberapa keuntungan signifikan:

   - **Konsistensi Lingkungan**

     Menghindari masalah "works on my machine" dengan

memastikan lingkungan yang identik di seluruh sistem pengembangan, staging, dan produksi.

- **Kemudahan Deployment**

  Mempermudah proses deployment dengan hanya menggunakan perintah **docker-compose up** untuk menjalankan seluruh stack aplikasi.
- **Skalabilitas**

  Memudahkan untuk meningkatkan kapasitas komponen individual, misalnya menambah lebih banyak backend containers berdasarkan kebutuhan.
- **Portabilitas**

  Memungkinkan aplikasi untuk dijalankan di berbagai platform (Windows, Linux, Mac, cloud) tanpa perlu modifikasi kode.
- **CI/CD Integration**

  Docker mempermudah integrasi dengan pipeline **CI/CD**, memungkinkan otomatisasi testing dan deployment.
- **Efisiensi Sumber Daya**

  Docker lebih ringan daripada virtualisasi tradisional, sehingga memungkinkan pengelolaan sumber daya yang lebih efisien.

Dengan penggunaan Docker, deployment aplikasi **Recipe Finder** menjadi lebih sederhana dan konsisten, sehingga mempermudah manajemen aplikasi dalam berbagai lingkungan.

Bab III

Analisis Pemecahan Masalah

3.1 Representasi Recipe Little Alchemy 2 sebagai Graph

1. Pemetaan Elemen sebagai Node

Dalam permainan **Little Alchemy 2**, elemen-elemen permainan dipetakan sebagai **node** dalam representasi graf. Setiap elemen, baik yang dasar maupun turunan, dianggap sebagai entitas graf yang memiliki sejumlah properti penting. Elemen dasar dalam permainan meliputi **Water**, **Fire**, **Earth**, **Air**, dan **Time**, yang dapat digunakan untuk membuat 720 elemen turunan melalui kombinasi.

Setiap node dalam graf ini memiliki beberapa atribut kunci:

- **ID Unik**:

  Setiap elemen memiliki nama atau ID yang berfungsi untuk mengidentifikasi elemen tersebut dalam sistem.

- **Tingkat Tier**

  Menunjukkan kompleksitas elemen. Elemen dasar berada di **Tier 0**, sementara elemen-elemen yang lebih kompleks, yang merupakan hasil kombinasi, berada di tier yang lebih tinggi.

- **Informasi Dasar/Turunan**

  Node mencatat apakah elemen tersebut merupakan elemen dasar atau hasil turunan dari kombinasi dua elemen.

- **URL Gambar**

  Untuk memberikan representasi visual elemen dalam aplikasi, masing-masing node dapat memiliki URL gambar yang menunjukkan simbol atau gambar elemen tersebut (opsional).

Dengan pemetaan ini, algoritma pencarian dapat menelusuri graf untuk mengidentifikasi elemen-elemen secara unik dan menganalisis hubungan antar elemen, memungkinkan pencarian kombinasi yang efektif.

2. Pemetaan Recipe sebagai Edge

Dalam graf **Little Alchemy 2**, hubungan antar elemen diwakili oleh **edge** yang menghubungkan dua node (elemen) melalui kombinasi. Setiap **recipe** (resep) menggambarkan sebuah edge yang menghubungkan dua

elemen bahan (ingredients) dan elemen hasil (result). Edge ini bersifat berarah, yang berarti kombinasi dua elemen menghasilkan satu elemen baru.

Secara formal, sebuah edge **E** antara dua elemen bahan (i_1, i_2) dan hasil (r) dapat didefinisikan sebagai:

$$\mathbf{E = (i_1, i_2) \rightarrow r}$$

di mana $\mathbf{i_1}$ dan $\mathbf{i_2}$ adalah elemen bahan, dan **r** adalah hasil dari kombinasi tersebut.

Dengan struktur ini, algoritma pencarian dapat mengeksplorasi berbagai kemungkinan kombinasi yang menghasilkan elemen tertentu, serta melacak elemen mana yang dapat dihasilkan dari kombinasi elemen lainnya.

3. **Hierarki Tier dalam Game**

Little Alchemy 2 menerapkan sistem **tier** untuk menandai kompleksitas setiap elemen, yang sangat penting dalam representasi graf. Sistem tier ini memisahkan elemen-elemen dalam graf berdasarkan tingkat kesulitan untuk membuatnya. Elemen-elemen dasar, seperti **Water**, **Fire**, **Earth**, dan **Air**, berada di **Tier 0**, sementara elemen-elemen yang lebih kompleks hasil dari kombinasi dua elemen berada di tier lebih tinggi.

Pentingnya sistem tier ini meliputi:

- **Tingkatan Kompleksitas**

  Tier 0 berisi elemen dasar, sementara elemen dengan tier yang lebih tinggi menggambarkan hasil kombinasi yang lebih rumit.

- **Batasan Kombinasi**

  Elemen dengan tier yang lebih tinggi hanya dapat dibentuk dari elemen dengan tier yang lebih rendah, menciptakan struktur **hierarkis** dalam graf. Ini memastikan bahwa pencarian hanya terjadi dalam urutan yang benar, contohnya kombinasi elemen hanya dapat dilakukan dengan cara yang sah.

- **Validasi Jalur**

  Dengan memanfaatkan tier, pencarian dapat diverifikasi untuk menghindari siklus atau kombinasi yang tidak valid, karena elemen lebih tinggi hanya bisa dibuat dari elemen yang lebih rendah.

Graf ini membentuk **layered graph**, yang di mana elemen-elemen lebih kompleks berada di lapisan atas dan dapat dihasilkan dengan

kombinasi elemen-elemen dari lapisan bawah. Hal ini juga mengarah pada **batasan pencarian** dalam algoritma, karena kombinasi hanya akan valid jika kedua elemen pembentuknya berada di tier yang lebih rendah daripada elemen hasil.

4. **Tantangan Siklus dalam Graph**

   Salah satu tantangan utama dalam merepresentasikan **Little Alchemy 2** sebagai graf adalah **kemungkinan terjadinya siklus**. Siklus dalam graf dapat muncul dalam beberapa kondisi:

   - **Kombinasi Rekursif**

     Di mana beberapa elemen dapat saling membentuk dirinya sendiri melalui kombinasi berulang.
   - **Alternatif Jalur**

     Banyak elemen dapat dibentuk melalui beberapa jalur kombinasi yang berbeda.
   - **Kombinasi Simetris**

     Dalam beberapa kasus, urutan kombinasi tidak penting (misalnya, kombinasi **A + B** menghasilkan elemen yang sama dengan **B + A**).

   Untuk mengatasi tantangan ini, beberapa teknik diterapkan dalam implementasi graf:

   - **Deteksi Siklus**

     Menggunakan struktur data seperti **visited set** untuk melacak elemen-elemen yang telah dikunjungi, sehingga siklus dapat dihindari.
   - **Batasan Tier**

     Memastikan bahwa hanya kombinasi yang menghasilkan elemen dengan tier lebih tinggi yang dipertimbangkan, mencegah eksplorasi jalur yang tidak sah.

3.2 Pemetaan Masalah menjadi Graph Search Problem

3.2.1 Problem Space

1. **Representasi State Space**

   State space dalam **Little Alchemy 2** mengacu pada semua kemungkinan keadaan yang dapat dicapai dalam permainan, yang terdiri dari semua elemen yang telah ditemukan dan kombinasi yang

telah dilakukan. Secara formal, state dapat didefinisikan sebagai **S = {E, D}**, di mana:

- **E** adalah himpunan elemen yang telah ditemukan.
- **D** adalah daftar kombinasi yang telah dilakukan.

Namun, karena banyaknya elemen dan kombinasi, state space ini sangat besar. Oleh karena itu, representasi state space kami dimodifikasi menjadi **S = {E_current, P}**, di mana:

- **E_current** adalah elemen yang sedang diproses.
- **P** adalah jalur atau path yang ditempuh untuk mencapai **E_current** dari elemen dasar.

Pendekatan ini memungkinkan pencarian yang lebih efisien dengan hanya menyimpan informasi yang relevan untuk merekonstruksi jalur menuju elemen target.

2. **Initial State (Elemen Dasar)**

   State awal dalam pencarian adalah himpunan elemen dasar yang belum dikombinasikan, yang diwakili sebagai:

   - **S_initial = {E_basic, ∅}**
   - **E_basic** = {Water, Fire, Earth, Air, Time}
   - ∅ menunjukkan bahwa belum ada kombinasi yang dilakukan.

   Elemen-elemen dasar ini berada di **Tier 0** dan merupakan titik awal untuk semua pencarian recipe. Semua elemen lain yang lebih kompleks harus dihasilkan dari kombinasi elemen-elemen dasar ini.

3. Goal State (Elemen Target)

   Goal state terjadi ketika algoritma pencarian berhasil menemukan elemen target yang diinginkan, yang dinyatakan sebagai:

   - **S_goal = {E_target, P}**
   - **E_target** adalah elemen target yang dicari.
   - **P** adalah jalur kombinasi yang digunakan untuk mencapai elemen target.

   Goal state ini tercapai ketika algoritma menemukan jalur dari elemen dasar yang menghasilkan elemen target.

4. **Langkah-langkah Transisi Antar State**

Transisi antar state dalam **Little Alchemy 2** terjadi melalui kombinasi dua elemen. Secara formal, transisi dapat dijelaskan sebagai:

$$T(S_1, (i_1, i_2)) = S_2$$

di mana:

- S_1 adalah state awal yang berisi elemen-elemen sebelumnya.
- (i_1, i_2) adalah pasangan elemen bahan.
- S_2 adalah state baru yang berisi elemen hasil dari kombinasi dan path yang telah ditempuh.

Untuk memastikan transisi yang valid, ada beberapa batasan yang harus diterapkan:

- Elemen i_1 dan i_2 harus ada dalam E_1.
- Kombinasi (i_1, i_2) harus valid dalam aturan permainan.
- **Tier** dari elemen hasil harus lebih tinggi daripada elemen bahan.

Karena sifat komutatif dari kombinasi ((i_1, i_2) sama dengan (i_2, i_1)), transisi ini dianggap setara, sehingga mengurangi redundansi dalam pencarian.

3.2.2 Implementasi Breadth-First Search

1. **Adaptasi BFS untuk Pencarian Recipe**

   Breadth-First Search (BFS) diimplementasikan untuk menelusuri graph kombinasi elemen secara **melebar**. Dimulai dari elemen-elemen dasar, BFS menjelajahi semua kemungkinan kombinasi level per level, sehingga menjamin ditemukan jalur terpendek menuju elemen target. BFS dimodifikasi dalam **Little Alchemy 2** sebagai berikut:

   - Memasukkan semua elemen dasar ke dalam **queue**.
   - Mengambil elemen dari queue dan mencoba semua kombinasi yang mungkin.
   - Menambahkan elemen hasil yang belum dikunjungi ke dalam queue.
   - Proses berlanjut hingga menemukan elemen target atau queue kosong.

Struktur data utama yang digunakan dalam implementasi BFS adalah:

- **Queue**

  Menyimpan elemen yang akan dieksplorasi.
- **Visited Map**

  Memetakan elemen yang telah dikunjungi untuk menghindari pengulangan.
- **Parent Map**

  Menyimpan informasi tentang elemen pembentuk dan recipe yang digunakan, memungkinkan rekonstruksi jalur saat target ditemukan.

BFS memastikan bahwa jalur terpendek ke elemen target ditemukan dengan mengeksplorasi elemen secara level demi level.

2. **Optimasi BFS untuk Problem Space Little Alchemy 2**

   Untuk meningkatkan efisiensi BFS dalam konteks **Little Alchemy 2**, beberapa teknik optimasi diterapkan yang membantu mengurangi ruang pencarian dan meningkatkan kinerja secara keseluruhan. Beberapa optimasi utama yang digunakan adalah sebagai berikut:

   - **Filtering Berdasarkan Tier**

     Hanya kombinasi yang menghasilkan elemen dengan **tier lebih tinggi** yang dipertimbangkan untuk dieksplorasi lebih lanjut. Hal ini mengurangi jumlah kombinasi yang perlu dicoba, karena kita tidak perlu mengeksplorasi jalur yang mengarah ke elemen yang tidak relevan atau yang memiliki tier yang lebih rendah.
   - **Pruning Berdasarkan Hierarki**

     Kombinasi yang tidak valid atau yang melanggar aturan hierarki tier, seperti mencoba menggabungkan dua elemen dengan tier yang lebih tinggi untuk menghasilkan elemen dengan tier lebih rendah, akan secara otomatis diabaikan oleh algoritma. Ini memastikan bahwa pencarian hanya terjadi dalam urutan yang benar dan sesuai dengan aturan permainan.

- **Caching Recipe**

  Setiap kombinasi yang menghasilkan elemen baru disimpan dalam cache, sehingga ketika kombinasi tersebut diperlukan lagi dalam pencarian yang berbeda, proses komputasi dapat dihindari. Caching membantu mengurangi beban komputasi dan mempercepat pencarian.

- **Early Termination**

  Begitu elemen target ditemukan, pencarian segera dihentikan tanpa perlu mengeksplorasi seluruh level atau semua kemungkinan dalam queue pada level yang sama. Ini memungkinkan algoritma untuk bekerja lebih cepat dalam menemukan solusi dan meminimalkan eksplorasi jalur yang tidak perlu.

Implementasi optimasi ini membuat BFS lebih efisien dalam mengatasi masalah besar dengan elemen dan kombinasi yang luas seperti dalam **Little Alchemy 2**, terutama untuk pencarian elemen dengan tier rendah hingga menengah.

3. **Rekonstruksi Jalur untuk Menemukan Recipe**

   Setelah elemen target ditemukan dalam pencarian BFS, langkah berikutnya adalah **rekonstruksi jalur** yang mengarah dari elemen dasar ke elemen target. Proses rekonstruksi ini menggunakan **parent map** yang dibangun selama proses pencarian.

   Langkah-langkah rekonstruksi:

   - **Mulai dari elemen target**

     Setelah menemukan elemen target, algoritma akan mulai melacak kembali jalur pencarian ke elemen dasar.

   - **Telusuri kembali ke parent**

     Dengan menggunakan parent map, algoritma mengikuti jejak langkah-langkah kombinasi yang digunakan untuk mencapai elemen target.

   - **Catat setiap kombinasi**

     Pada setiap langkah, catat kombinasi dua elemen yang digunakan untuk menghasilkan elemen yang lebih tinggi. Hal ini

memungkinkan kita untuk merekonstruksi setiap langkah pencarian secara urut.

- **Balik urutan langkah**

  Setelah mencapai elemen dasar, jalur yang ditemukan dibalik urutannya untuk mendapatkan urutan langkah dari elemen dasar ke target.

Proses rekonstruksi ini memungkinkan pemain atau aplikasi untuk memvisualisasikan langkah-langkah yang harus diambil untuk mencapai elemen target dari elemen-elemen dasar.

3.2.3 Implementasi Depth-First Search

1. **Adaptasi DFS untuk Pencarian Recipe**

   Depth-First Search (DFS) digunakan untuk menelusuri graf secara mendalam, yaitu mengeksplorasi satu jalur kombinasi sedalam mungkin sebelum kembali dan mencoba jalur lain. Dalam implementasi untuk **Little Alchemy 2**, pendekatan DFS dilakukan dengan menggunakan rekursi untuk mengeksplorasi kombinasi elemen-elemen secara mendalam, hingga menemukan elemen target atau mencapai batas kedalaman yang ditentukan.

   Untuk menghindari pencarian yang terlalu dalam atau eksplorasi yang tidak perlu, DFS dalam konteks ini dilengkapi dengan **batas kedalaman**:

   - **Dimulai dari elemen dasar**.
   - **Secara rekursif menelusuri kombinasi** untuk menghasilkan elemen baru.
   - **Menerapkan batas kedalaman** untuk menghindari eksplorasi yang berlebihan, sehingga pencarian tetap efisien.
   - **Backtracking** dilakukan ketika batas kedalaman tercapai atau ketika elemen target ditemukan.

2. **Struktur Kode Implementasi DFS**

   Beberapa struktur data yang digunakan dalam implementasi DFS adalah sebagai berikut:

   - **Call Stack (rekursif)**

     Setiap panggilan fungsi DFS menambah kedalaman pencarian,

dan ketika sebuah jalur selesai, DFS akan melakukan backtrack untuk mencoba jalur lain.

- **Visited Map**

  Digunakan untuk melacak elemen-elemen yang sudah dikunjungi selama pencarian untuk menghindari eksplorasi jalur yang berulang.

- **Parent Map**

  Menyimpan informasi tentang elemen pembentuk dan recipe yang digunakan, memungkinkan rekonstruksi jalur.

- **Depth Counter**

  Variabel yang melacak kedalaman pencarian saat ini dalam pencarian rekursif.

Keuntungan utama dari pendekatan rekursif ini adalah kemudahan implementasi, tetapi ada risiko **stack overflow** untuk pencarian yang sangat dalam, sehingga batas kedalaman sangat penting untuk diterapkan.

3. **Optimasi DFS untuk Problem Space Little Alchemy 2**

   Beberapa teknik optimasi penting diterapkan dalam DFS untuk meningkatkan efisiensinya dalam **Little Alchemy 2**, termasuk:

   - **Depth Limiting**

     Membatasi kedalaman pencarian berdasarkan **tier** elemen target untuk menghindari eksplorasi yang tidak perlu. Ini mencegah DFS mencoba jalur yang lebih dalam dari yang diperlukan untuk menemukan solusi.

   - **Prioritization**

     Memprioritaskan eksplorasi kombinasi yang menghasilkan elemen dengan **tier lebih tinggi** terlebih dahulu. Hal ini mengoptimalkan jalur pencarian dengan mengarah pada elemen-elemen yang lebih kompleks lebih cepat.

   - **Memoization**

     Menyimpan hasil pencarian untuk **subproblem** yang telah dieksplorasi sebelumnya. Dengan demikian, jika subproblem

yang sama ditemukan lagi, pencarian dapat dilanjutkan dari hasil yang sudah ada tanpa perlu menghitung ulang.

- **Iterative Deepening**

  Untuk pencarian jalur yang lebih banyak, teknik **iterative deepening** digunakan untuk mencari beberapa jalur alternatif dengan menambah kedalaman pencarian secara bertahap.

- **Randomization**

  Menambahkan elemen keacakan pada urutan eksplorasi untuk menemukan jalur alternatif yang bervariasi. Ini berguna untuk menemukan berbagai jalur dalam graf yang lebih besar dan kompleks.

4. **Rekonstruksi Jalur untuk Menemukan Recipe**

   Proses rekonstruksi jalur dalam DFS mirip dengan BFS, menggunakan **parent map** yang dibangun selama pencarian. Rekonstruksi ini akan menghasilkan rangkaian recipe yang dapat diikuti untuk membuat elemen target dari elemen dasar.

   Namun, perlu dicatat bahwa DFS tidak selalu menjamin jalur terpendek, karena DFS cenderung menemukan jalur yang lebih dalam terlebih dahulu, yang mungkin lebih panjang dibandingkan jalur BFS. Meski demikian, DFS dapat menemukan **salah satu jalur valid** untuk mencapai elemen target.

3.2.4 Implementasi Bidirectional Search

1. Forward Search dari Elemen Dasar

   Forward search adalah bagian pertama dari **Bidirectional Search**, yang bekerja dengan menelusuri graf dari arah **elemen dasar** menuju **elemen target**. Proses ini dimulai dengan memasukkan semua elemen dasar ke dalam **forward queue** dan mengeksplorasi kombinasi yang dapat dihasilkan.

   Langkah-langkah forward search:

- Mulai dengan memasukkan elemen dasar ke dalam forward queue.

- Eksplorasi kombinasi yang menghasilkan elemen baru, menuju **tier yang lebih tinggi**.
- Menjaga **visited set** untuk melacak elemen yang sudah dijelajahi.

2. **Backward Search dari Elemen Target**

   Backward search adalah komponen kedua dari Bidirectional Search yang bekerja dengan menelusuri graf dari **elemen target** kembali ke **elemen dasar**. Ini dimulai dengan memasukkan elemen target ke dalam **backward queue** dan mencoba menemukan elemen yang membentuknya.

   Langkah-langkah backward search:

   - Memasukkan elemen target ke dalam backward queue.
   - Eksplorasi elemen pembentuk dari target, dengan menelusuri elemen-elemen yang dapat digunakan untuk membentuk target.
   - Menjaga **visited set** untuk melacak elemen yang sudah dijelajahi selama pencarian mundur.

3. **Titik Pertemuan dan Rekonstruksi Jalur Lengkap**

   Saat kedua pencarian (forward dan backward) bertemu di titik yang sama, **titik pertemuan** ini menunjukkan bahwa jalur dari elemen dasar ke elemen target telah ditemukan. Setelah titik pertemuan ditemukan, jalur lengkap dapat direkonstruksi dengan menggabungkan jalur dari kedua arah:

   - **Jalur Forward**

     Dibangun dengan mengikuti **parent map** dari titik pertemuan ke elemen dasar.
   - **Jalur Backward**

     Dibangun dengan mengikuti **child map** dari titik pertemuan ke elemen target.

   Dengan menggabungkan kedua jalur tersebut, kita mendapatkan jalur lengkap dari elemen dasar menuju elemen target.

4. **Penanganan Validasi Jalur Berdasarkan Tier**

Validasi jalur berdasarkan **tier** sangat penting dalam **Bidirectional Search** untuk memastikan bahwa:

- Semua elemen **ingredient** memiliki tier yang lebih rendah dari elemen **hasil**.
- Forward search hanya bergerak menuju **tier yang lebih tinggi**, dan backward search hanya bergerak menuju **tier yang lebih rendah**.

Validasi ini mengurangi jumlah kombinasi yang perlu dieksplorasi dan memastikan bahwa jalur yang direkonstruksi adalah **jalur valid** dalam konteks aturan permainan **Little Alchemy 2**.

3.3 Arsitektur Aplikasi

Aplikasi **Recipe Finder** untuk **Little Alchemy 2** mengadopsi arsitektur **client-server** dengan pemisahan frontend dan backend yang jelas. Arsitektur ini memberikan keuntungan dalam hal modularitas, skalabilitas, dan kemudahan pengelolaan, serta memastikan pengalaman pengguna yang optimal. Dalam pengembangan aplikasi ini, **Next.js** digunakan untuk frontend dan **Go (Golang)** untuk backend. Komunikasi antara frontend dan backend dilakukan melalui **REST API** yang efisien. Berikut adalah penjelasan lebih lanjut tentang struktur dan aliran data dalam aplikasi.

3.3.1 Diagram Arsitektur Aplikasi

Pada aplikasi **Recipe Finder**, frontend dan backend dipisah secara jelas, dengan frontend yang bertanggung jawab untuk tampilan dan interaksi pengguna, sementara backend menangani logika pencarian dan pemrosesan data. Diagram berikut menggambarkan aliran komunikasi antar komponen:

1. **Frontend (Next.js)**
    - **UI/UX**

      Mengelola antarmuka pengguna dengan komponen seperti form pencarian, pemilihan algoritma, visualisasi tree, dan statistik pencarian.

- **State Management**

  Mengelola state aplikasi, termasuk elemen yang sudah ditemukan dan jalur yang sudah diproses.
- **API Request**

  Mengirim request pencarian ke backend untuk memulai pencarian elemen berdasarkan parameter yang dimasukkan pengguna.

2. **Backend (Go/Golang)**
   - **REST API**

     Menangani request dari frontend dan mengembalikan hasil pencarian dalam format JSON.
   - **Algoritma Pencarian**

     Menangani pencarian menggunakan algoritma BFS, DFS, atau Bidirectional Search, dan menghitung jalur kombinasi.
   - **Data**

     Mengelola file data elemen dan resep (misalnya elements.json) serta memproses hasil pencarian.
3. **Komunikasi Antar-Komponen**
   - **Frontend → Backend**

     Frontend mengirim HTTP POST request ke backend melalui API untuk memulai pencarian elemen target.
   - **Backend → Frontend**

     Setelah pemrosesan selesai, backend mengembalikan hasil pencarian (jalur, statistik, visualisasi) dalam format JSON ke frontend.

3.3.2 Aliran Data Antar Komponen

Aliran data dalam aplikasi mengikuti pola **request-response** dengan beberapa tahapan. Berikut adalah detail aliran data antara frontend dan backend:

1. Input Pengguna ke Frontend
   - Pengguna memasukkan elemen target yang ingin dicari.
   - Pengguna memilih algoritma pencarian yang diinginkan (BFS, DFS, atau Bidirectional Search).

- Pengguna memilih mode pencarian (single atau multiple recipe) dan dapat menetapkan parameter tambahan seperti jumlah maksimal recipe yang ingin dicari.

2. **Frontend ke Backend**
    - Frontend membuat **HTTP POST request** ke endpoint API backend, mengirimkan parameter pencarian seperti target elemen, algoritma yang dipilih, dan mode pencarian.
    - Data dikirim dalam format JSON untuk mempermudah pemrosesan di sisi backend.
3. **Pemrosesan di Backend**
    - Backend menerima request, melakukan validasi parameter yang dikirim dari frontend, dan memastikan bahwa data elemen dan resep yang diperlukan tersedia.
    - Backend menjalankan algoritma pencarian yang dipilih (BFS, DFS, atau Bidirectional Search), memulai pencarian untuk menemukan jalur menuju elemen target.
    - Backend menghasilkan jalur kombinasi yang diperlukan untuk membuat elemen target dan menghasilkan **visualisasi tree** yang menggambarkan langkah-langkah kombinasi elemen.
4. **Backend ke Frontend**
    - Backend mengirim hasil pencarian kembali ke frontend dalam format **JSON**, yang mencakup:
        - Jalur (path) yang ditemukan.
        - Statistik pencarian (misalnya waktu eksekusi dan jumlah node yang dikunjungi).
        - Struktur visualisasi tree yang menggambarkan bagaimana elemen-elemen digabungkan.
    - Frontend menerima data dan memprosesnya untuk menampilkan visualisasi dan informasi yang relevan.
5. **Rendering di Frontend**
    - Frontend menerima hasil pencarian dan memperbarui state aplikasi.

- Jalur kombinasi elemen ditampilkan sebagai daftar berurutan yang mudah dibaca oleh pengguna.
- Menggunakan **ReactFlow**, frontend membangun **visualisasi tree** yang interaktif, memudahkan pengguna untuk melihat hubungan antar elemen.
- Statistik pencarian, seperti waktu eksekusi dan jumlah node yang dikunjungi, juga ditampilkan untuk memberi informasi lebih lanjut kepada pengguna.

3.3.3 Penanganan Request dan Response

Penanganan request dan response dalam aplikasi sangat penting untuk memastikan data dikirim dengan benar antara frontend dan backend. Berikut adalah komponen-komponen utama dalam penanganan request dan response:

1. **Request Handler Frontend**
   - Mengumpulkan data input dari pengguna dan mengonversinya menjadi payload **JSON**.
   - Mengirim request menggunakan **fetch API** dengan metode POST untuk berkomunikasi dengan backend.
   - Menampilkan **indikator loading** untuk memberi tahu pengguna bahwa request sedang diproses.
   - Menangani error dan timeout jika terjadi masalah dalam komunikasi dengan backend.
2. **API Gateway**
   - API route di Next.js bertindak sebagai proxy antara frontend dan backend.
   - Menyederhanakan penanganan **CORS** dan **error** sehingga komunikasi antara keduanya lebih mudah.
   - Meneruskan parameter pencarian ke backend dan mengolah respons yang diterima.
3. **Backend Request Handler**
   - Menerima parameter dari frontend dalam bentuk **HTTP request** dan melakukan validasi input.

- Memeriksa apakah data elemen dan resep tersedia di backend (misalnya dalam file elements.json).
- Mengarahkan request ke algoritma pencarian yang sesuai berdasarkan parameter yang diterima.

4. **Response Parsing**
    - Backend menghasilkan output yang terstruktur berupa jalur recipe, statistik, dan visualisasi tree.
    - Frontend mem-parsing output ini menjadi struktur data yang dapat digunakan untuk tampilan antarmuka pengguna.
    - Frontend memproses **tree structure** untuk visualisasi interaktif dan menampilkan jalur pencarian yang ditemukan.

3.3.4 Visualisasi Hasil Pencarian

Salah satu fitur utama aplikasi adalah kemampuan untuk **memvisualisasikan** hasil pencarian dalam bentuk tree, yang memudahkan pengguna untuk melihat bagaimana elemen-elemen digabungkan untuk mencapai elemen target. Berikut adalah beberapa cara visualisasi hasil pencarian dalam aplikasi:

1. **Recipe Path Display**
    - Menampilkan langkah-langkah pembuatan elemen target sebagai daftar berurutan.
    - Menyoroti elemen **ingredient** dan **result** pada setiap langkah, sehingga pengguna dapat mengikuti proses pembuatan.
    - Menunjukkan **tier** dari setiap elemen untuk memberikan konteks mengenai tingkat kompleksitasnya.
2. **Tree Visualization**
    - Menggunakan **ReactFlow** untuk membangun visualisasi interaktif dari recipe tree.
    - Elemen target ditempatkan di bagian atas tree, sementara elemen dasar (tier 0) berada di bagian bawah.
    - **Edge** yang menghubungkan node menggambarkan kombinasi elemen yang menghasilkan elemen target.
    - Pengguna dapat melakukan **zoom**, **pan**, dan **navigasi** untuk menjelajahi tree yang lebih besar dan kompleks.

3. **Search Statistics**
    - Menampilkan statistik terkait pencarian, seperti **waktu eksekusi** dan **jumlah node yang dikunjungi** selama pencarian.
    - Menyediakan perbandingan statistik untuk berbagai algoritma pencarian yang digunakan, seperti BFS, DFS, dan Bidirectional Search.
4. **Node Styling**
    - Elemen target diberi **warna hijau** dengan **border tebal** untuk menandai elemen yang dicari.
    - Elemen dasar (tier 0) diberi **warna biru** dengan **border tebal** untuk membedakannya dari elemen lain.
    - Elemen perantara diberi **warna abu-abu** untuk menandakan bahwa elemen tersebut merupakan bagian dari jalur pencarian, tetapi bukan elemen target atau dasar.
    - Node **"combining"** khusus menunjukkan titik kombinasi elemen yang digunakan untuk menghasilkan elemen lebih lanjut.

3.4 Contoh Ilustrasi Kasus

Contoh ilustrasi kasus ini menggambarkan penggunaan algoritma pencarian untuk menemukan recipe untuk elemen-elemen dalam **Little Alchemy 2**.

3.4.1 Contoh Kasus 1: Pencarian Recipe untuk Elemen Sederhana (Metal)

Elemen **Metal** adalah elemen yang relatif sederhana dan terletak di **Tier 2**. Pencarian menggunakan berbagai algoritma akan menunjukkan jalur yang berbeda, tetapi tetap menghasilkan elemen yang sama.

1. **Hasil Pencarian BFS**
    - Dimulai dari elemen dasar (Water, Fire, Earth, Air).
    - Mengeksplorasi kombinasi pertama **Earth + Fire → Metal**.
    - Metal ditemukan pada level pertama eksplorasi.
2. **Hasil Pencarian giDFS**
    - Dimulai dengan elemen Fire.
    - Mengeksplorasi kombinasi Fire + Earth → Metal.

- Menemukan Metal dengan 4 node yang dikunjungi.

3. **Hasil Pencarian Bidirectional**
    - Forward search dari elemen dasar.
    - Backward search dari Metal.
    - Titik pertemuan pada Fire.
    - Jalur ditemukan dengan 5 node yang dikunjungi.
4. **Analisis**
    - Semua algoritma menemukan jalur terpendek karena Metal berada pada tier rendah.
    - DFS sedikit lebih cepat karena lebih langsung menemukan solusi.
    - Bidirectional Search mengurangi jumlah node yang dikunjungi dibandingkan dengan BFS.

3.4.2 Contoh Kasus 2: Pencarian Recipe untuk Elemen Kompleks (Picnic)

1. **Hasil Pencarian BFS**
    - Mengeksplorasi level per level dan menemukan jalur dengan total 6 kombinasi.
    - Jumlah node yang dikunjungi: 278.
    - Waktu eksekusi: 156ms.
2. **Hasil Pencarian DFS**
    - Mengeksplorasi jalur secara mendalam, menemukan jalur dengan 8 kombinasi.
    - Jumlah node yang dikunjungi: 187.
    - Waktu eksekusi: 98ms.
3. **Hasil Pencarian Bidirectional**
    - Forward search dan backward search bertemu di elemen Human.
    - Jumlah node yang dikunjungi: 143.
    - Waktu eksekusi: 87ms.
4. **Analisis**
    - **BFS** menemukan jalur terpendek dengan 6 langkah.
    - **DFS** menemukan jalur alternatif yang lebih panjang.

- **Bidirectional Search** lebih efisien dalam hal jumlah node yang dikunjungi dan waktu eksekusi.

5. **Visualisasi**

   Picnic memiliki visualisasi tree yang lebih kompleks dengan banyak percabangan yang terjadi pada level yang lebih tinggi.

Bab IV
Implementasi dan Pengujian

4.1 Spesifikasi Teknis Program

4.1.1 Struktur Data

Pada implementasi aplikasi **Recipe Finder** untuk **Little Alchemy 2**, struktur data yang digunakan sangat krusial untuk memastikan efisiensi dalam pencarian dan visualisasi. Setiap komponen dalam aplikasi, baik di frontend maupun backend, memanfaatkan struktur data yang sesuai untuk menyederhanakan pemrosesan elemen-elemen dan recipe dalam permainan.

1. **Representasi Elemen dan Recipe**

   Elemen-elemen dalam **Little Alchemy 2** dipetakan menggunakan struktur **Element**, yang menyimpan informasi dasar mengenai elemen seperti ID (nama elemen), daftar recipe yang menghubungkan elemen bahan dengan elemen hasil, dan URL gambar untuk representasi visual elemen tersebut. **ElementStore** berfungsi sebagai **in-memory database** untuk menyimpan seluruh elemen dan recipe, termasuk memetakan setiap elemen ke dalam **tier** tertentu. Hal ini memungkinkan program untuk memproses elemen dengan lebih efisien dan mendukung pencarian yang lebih cepat.

   Data elemen dan recipe dimuat dari file **JSON** (elements.json) yang berisi data hasil scraping dari situs **Little Alchemy 2**, yang mencakup lebih dari 720 elemen dan kombinasi resepnya. Struktur ini mendukung operasi pencarian yang efisien dengan kompleksitas **O(1)** untuk pencarian elemen berdasarkan **ID** dan **tier**, memastikan bahwa pencarian dapat dilakukan dengan cepat meskipun jumlah elemen sangat besar.

2. **Struktur Data untuk Pencarian Graph**

   Algoritma pencarian menggunakan berbagai struktur data penting untuk mempercepat proses eksplorasi graf. Berikut adalah beberapa struktur yang digunakan:

- **Queue**

  Pencarian menggunakan **queue** yang dikelola dengan menggunakan container/list dari Go's standard library. Queue memungkinkan BFS untuk menelusuri graf secara **level-by-level**, menjaga urutan eksplorasi sesuai dengan tingkatan kedalaman.
- **Recursive Stack**

  DFS menggunakan **recursive stack** untuk eksplorasi mendalam, di mana setiap panggilan rekursif menambah kedalaman pencarian. Stack ini memudahkan DFS untuk melacak jalur yang sedang dieksplorasi, serta menghindari pencarian yang berlebihan.
- **Visited Sets**

  Struktur **visited set** yang diimplementasikan dengan **map[string]bool** digunakan untuk melacak elemen yang sudah dikunjungi, mencegah eksplorasi berulang dan membantu mendeteksi siklus dalam graf.
- **Parent Maps**

  Untuk memungkinkan rekonstruksi jalur setelah pencarian selesai, struktur **parent map** menyimpan informasi mengenai hubungan antara parent dan child elemen yang terhubung dalam proses pencarian.
- **Tier Maps**

  Map ini digunakan untuk melacak tier masing-masing elemen dalam graf, memastikan bahwa kombinasi elemen yang terjadi menghormati hierarki yang telah ditentukan, di mana elemen dengan tier lebih tinggi hanya bisa dibentuk dari elemen dengan tier lebih rendah.

3. **Tree Structure untuk Visualisasi**

   Visualisasi hasil pencarian menggunakan **ReactFlow** di frontend memerlukan struktur tree yang dioptimalkan. Pada frontend, tree structure ini diterjemahkan menjadi komponen **ReactFlow**

dengan berbagai properti yang memungkinkan representasi visual interaktif dari kombinasi elemen.

- **Tree Structure**

  Struktur tree ini mendukung visualisasi hierarkis, dengan elemen target berada di bagian atas dan elemen dasar di bagian bawah. Setiap node dalam tree mewakili sebuah elemen, dan **edge** yang menghubungkan node menunjukkan bagaimana elemen-elemen tersebut dapat digabungkan.

- **Styling dan Highlighting**

  Node di dalam tree dapat dibedakan berdasarkan tipenya, seperti elemen target, elemen dasar, dan elemen bahan, dengan menggunakan styling yang berbeda. Setiap elemen juga memuat informasi **tier** yang memudahkan pengguna untuk memahami level kompleksitas elemen tersebut.

- **Cycle Detection**

  Salah satu tantangan utama dalam visualisasi adalah menangani siklus dalam graf. Dengan menggunakan struktur tree ini, aplikasi mampu mendeteksi dan memvisualisasikan siklus yang mungkin terjadi dalam kombinasi elemen.

4.1.2 Komponen Frontend

Komponen **frontend** aplikasi **Recipe Finder** dibangun menggunakan **Next.js** dan **React** untuk memberikan pengalaman pengguna yang interaktif dan dinamis. Beberapa komponen utama di frontend meliputi:

1. **Komponen Pencarian dan Input**

   Frontend menyediakan berbagai komponen untuk memudahkan pengguna dalam memasukkan elemen target dan memilih algoritma yang sesuai. Beberapa fitur utama komponen pencarian antara lain:

2. **Komponen Visualisasi Recipe Path**

   Komponen ini menampilkan jalur pembuatan elemen target dalam format **step-by-step** yang mudah dipahami, memungkinkan pengguna melihat proses secara berurutan.

3. **Komponen Visualisasi Tree**

   Komponen **RecipeTree** menggunakan **ReactFlow** untuk membangun **tree visualization** interaktif, yang menampilkan hubungan antar elemen dengan cara yang lebih visual dan intuitif. Dengan kontrol zoom, pan, dan MiniMap, pengguna dapat mengeksplorasi tree yang lebih besar dengan mudah.

4. **State Management dan Komunikasi dengan Backend**

   Di sisi frontend, state aplikasi dikelola menggunakan **React hooks** seperti **useState**, **useEffect**, dan **useCallback**. Komunikasi dengan backend dilakukan secara **asynchronous** menggunakan **fetch API**, di mana respons dari backend berupa **JSON** digunakan untuk memperbarui UI secara real-time.

4.1.3 Fungsi Utama Backend

Backend aplikasi **Recipe Finder** bertanggung jawab untuk melakukan pencarian algoritmik menggunakan tiga metode utama, yaitu **BFS**, **DFS**, dan **Bidirectional Search**, untuk menemukan jalur pembuatan elemen target dari elemen dasar. Setiap algoritma memiliki modifikasi dan optimasi yang disesuaikan dengan kebutuhan permainan **Little Alchemy 2**.

1. **Implementasi Algoritma BFS**

   Algoritma **Breadth-First Search (BFS)** diimplementasikan dalam file **bfs.go**. Fungsi utama yang digunakan adalah **FindShortestPath**, yang melakukan pencarian untuk menemukan jalur terpendek dari elemen dasar ke elemen target. Beberapa optimasi yang diterapkan di dalam BFS adalah:

   - **Batasan Tier**

     Fungsi **getPossibleRecipesThatRespectTiers** memastikan bahwa hanya kombinasi yang menghasilkan elemen dengan **tier lebih tinggi** yang dipertimbangkan, memfilter jalur yang tidak valid.

   - **Early Termination**

     Pencarian dihentikan segera setelah elemen target ditemukan,

mengurangi jumlah eksplorasi yang tidak perlu meskipun masih ada elemen dalam queue pada level yang sama.

- **Memory Management**

  Pengelolaan **visited set** dan **parent map** dioptimalkan untuk efisiensi penggunaan memori, memastikan bahwa pencarian berjalan lancar tanpa menyebabkan **memory leak**.

- **Pruning**

  Proses pruning diterapkan untuk menghindari jalur yang tidak mengarah pada peningkatan tier, yang membantu mengurangi ruang pencarian secara signifikan.

2. **Implementasi Algoritma DFS**

   Algoritma **Depth-First Search (DFS)** diimplementasikan dalam file **dfs.go** dan menggunakan pendekatan **rekursif**. Fungsi utama dalam DFS adalah **FindShortestPath** yang menjadi titik masuk, sementara **dfsSearchWithTiers** melakukan pencarian secara mendalam. Beberapa optimasi yang diterapkan adalah:

   - **Depth Limiting**

     Kedalaman pencarian dibatasi sesuai dengan **tier target** untuk mencegah eksplorasi berlebihan yang dapat menyebabkan stack overflow.

   - **Tier-Based Pruning**

     Hanya kombinasi yang menghasilkan elemen dengan **tier lebih tinggi** yang dipertimbangkan, mirip dengan apa yang diterapkan dalam BFS.

   - **Backtracking**

     Backtracking dilakukan dengan menghapus elemen dari visited set dan parent map jika jalur yang dipilih tidak mengarah pada target, memungkinkan pencarian untuk melanjutkan dengan jalur lain.

   - **Starting Point Variation**

     Untuk meningkatkan kemungkinan menemukan jalur yang lebih optimal, pencarian mencoba semua elemen dasar sebagai titik awal.

3. **Implementasi Algoritma Bidirectional Search**

   Bidirectional Search menggabungkan pencarian maju (dari elemen dasar) dan mundur (dari elemen target) untuk menemukan jalur dengan lebih efisien. Implementasi ini ada dalam file **bid.go** dan memiliki fitur-fitur berikut:

   - **Backward Search Optimization**

     Pencarian mundur dioptimalkan dengan menggunakan indeks **recipe** berdasarkan hasil untuk mempercepat pencarian elemen pembentuk.
   - **Level-by-Level Expansion**

     Pencarian maju dan mundur dilakukan dengan pendekatan **level-by-level**, menjaga keseimbangan eksplorasi kedua arah agar proses pencarian lebih efisien.
   - **Tier Validation**

     Validasi yang ketat terhadap tier di kedua arah pencarian untuk memastikan kombinasi yang ditemukan sah.
   - **Path Reconstruction**

     Setelah titik pertemuan ditemukan, jalur dikonstruksi dengan menggabungkan jalur dari kedua arah pencarian.
   - **Early Termination**

     Proses dihentikan segera setelah kedua arah pencarian bertemu di titik pertemuan, menghemat waktu pencarian.

4. **Endpoint API dan Penanganan Request**

   Backend menyediakan **API endpoint** yang diimplementasikan melalui **Next.js API Routes**, yang menghubungkan frontend dengan program Go. Endpoint utama adalah **route.ts**, yang menangani komunikasi antara frontend dan backend. Fungsi **parseTreeStructure** di backend mengkonversi output dari CLI Go menjadi struktur data **JSON**, yang kemudian

5. **Multithreading untuk Optimasi Pencarian Multiple Recipe**

   Untuk optimasi pencarian **multiple recipe paths**, backend menggunakan **goroutines** di Go untuk memanfaatkan paralelisme dan

mempercepat pencarian. Beberapa teknik yang digunakan dalam implementasi multithreading meliputi:

- **Goroutines**
  Pencarian dijalankan dalam beberapa thread ringan (goroutines) untuk mengoptimalkan waktu pencarian.
- **Semaphores**
  Semaphores digunakan untuk membatasi jumlah goroutines yang aktif bersamaan, mencegah **resource exhaustion**.
- **WaitGroups**
  Sinkronisasi dilakukan menggunakan **WaitGroups** untuk memastikan bahwa semua goroutines selesai sebelum melanjutkan.
- **Mutex**
  Mutex digunakan untuk mengamankan akses ke shared resources, seperti hasil pencarian dan **path signature map**, untuk mencegah kondisi race.

4.2 Tata Cara Penggunaan Program

1. Instalasi dan Setup

   Program **Recipe Finder** dapat dijalankan baik secara **lokal** maupun menggunakan **Docker**. Berikut adalah panduan instalasi untuk kedua metode:

   - Instalasi Lokal

     1) **Prerequisites**
        Pastikan memiliki **Node.js v14+**, **Go v1.18+**, dan **Git** terpasang.
     2) **Clone Repository**
        Clone repository dari GitHub.
     3) **Setup Backend**
        Install dependensi Go dan pastikan file elements.json tersedia.
     4) **Setup Frontend**
        Install dependensi npm menggunakan npm install.
     5) **Menjalankan Backend**
        Jalankan backend menggunakan go run.

6) **Menjalankan Frontend**

   Jalankan frontend menggunakan npm run dev.

7) **Akses Aplikasi**

   Buka browser dan akses http://localhost:3000.

- Instalasi dengan Docker

1) **Prerequisites**

   Install **Docker** dan **Docker Compose**.

2) **Clone Repository**

   Clone repository dari GitHub.

3) **Build dan Jalankan dengan Docker Compose**

   Gunakan perintah docker-compose up --build untuk membangun dan menjalankan aplikasi.

4) **Akses Aplikasi**

   Akses aplikasi melalui http://localhost:3000.

2. Antarmuka Utama dan Komponen-Komponen
3. Langkah-Langkah Pencarian Recipe

   Pencarian recipe dalam aplikasi mengikuti langkah-langkah yang terstruktur untuk memastikan proses yang intuitif dan mudah dipahami oleh pengguna. Berikut adalah langkah-langkah detail untuk melakukan pencarian recipe:

   - **Masukkan Elemen Target**
     1) Ketik nama elemen yang ingin dicari dalam **input field**.
     2) Fitur **autocomplete** akan mengusulkan elemen yang relevan saat pengguna mengetik, sehingga mempercepat pencarian dan meminimalkan kesalahan input.
     3) **Validasi real-time** memeriksa apakah elemen yang dimasukkan valid, memberi feedback langsung kepada pengguna.
   - **Pilih Algoritma Pencarian**

     Pengguna dapat memilih salah satu dari tiga algoritma pencarian:

     1) **BFS (Breadth-First Search)**

        Digunakan untuk menemukan jalur terpendek, sangat cocok untuk elemen dengan **tier rendah**.

2) **DFS (Depth-First Search)**

   Memberikan eksplorasi yang lebih mendalam, lebih cepat untuk menemukan solusi tetapi tidak selalu optimal.

3) **Bidirectional Search**

   Efisien untuk elemen dengan **tier tinggi**, mengurangi ruang pencarian dengan pencarian maju dan mundur secara bersamaan.

- **Pilih Mode Pencarian**

  1) **Single Recipe Mode**

     Mode ini mencari **satu jalur** optimal untuk membuat elemen target.

  2) **Multiple Recipe Mode**

     Mode ini memungkinkan pencarian untuk menemukan **beberapa jalur** yang berbeda menuju elemen target. Pengguna juga dapat menentukan **jumlah maksimum recipe** yang ingin ditemukan.

- **Konfigurasi Multiple Recipe (Opsional)**

  1) Jika mode **Multiple Recipe** dipilih, pengguna dapat menentukan **jumlah maksimum recipe** yang ingin dicari, misalnya 3, 5, atau lebih.
  2) Nilai default adalah 3, dan disarankan untuk memilih antara 5 hingga 10 untuk elemen-elemen kompleks.

- **Mulai Pencarian**

  1) Klik tombol **"Cari Recipe"** untuk memulai pencarian.
  2) Indikator **loading** akan muncul selama proses pencarian berlangsung, memberi tahu pengguna bahwa aplikasi sedang bekerja.

- **Lihat Hasil**

  Setelah pencarian selesai, hasil akan ditampilkan dalam dua format:

  1) **Recipe Path**

     Daftar langkah-langkah kombinasi elemen dalam format teks yang mudah dipahami.

  2) **Tree Visualization**

     Representasi visual dari struktur recipe yang menunjukkan hubungan antar elemen dalam bentuk tree.

- **Eksplorasi Tree (Opsional)**

1) Gunakan kontrol **zoom** dan **pan** untuk menjelajahi tree.
2) MiniMap memungkinkan navigasi cepat pada tree yang besar, memudahkan pengguna untuk berpindah antar bagian tree tanpa kehilangan konteks.

- **Coba Pencarian Lain (Opsional)**
  1) Pengguna dapat mengubah elemen target, algoritma pencarian, atau mode pencarian untuk mencoba pencarian baru.
  2) Perbandingan hasil dari berbagai algoritma dapat membantu pengguna memilih jalur yang paling efisien atau optimal untuk elemen yang dicari.

4. Pemilihan Algoritma dan Mode

Pemilihan algoritma dan mode pencarian yang tepat sangat penting untuk mendapatkan hasil yang optimal dan efisien. Berikut adalah panduan untuk memilih algoritma dan mode yang sesuai dengan elemen yang ingin dicari.

- **Pemilihan Algoritma**
  1) **Breadth-First Search (BFS)**
     - **Kapan digunakan**
       Untuk menemukan **jalur terpendek** dengan langkah minimum.
     - **Kelebihan**
       Menjamin menemukan jalur terpendek, cocok untuk elemen dengan **tier rendah**.
     - **Kekurangan**
       Memerlukan **memori lebih besar** untuk elemen yang lebih kompleks.
     - **Cocok untuk**
       Elemen dengan **tier rendah hingga menengah** (tier 0–5).
  2) **Depth-First Search (DFS)**
     - **Kapan digunakan**
       Untuk pencarian **mendalam** atau eksplorasi jalur alternatif

- **Kelebihan**

  Penggunaan **memori lebih efisien**, lebih cepat untuk menemukan solusi meskipun tidak selalu optimal.
- **Kekurangan**

  Tidak menjamin jalur terpendek, bisa **terjebak dalam jalur yang sangat dalam**.
- **Cocok untuk**

  Eksplorasi **variasi jalur** atau elemen dengan jalur dalam.

3) **Bidirectional Search**
   - **Kapan digunakan**

     Untuk elemen yang lebih **kompleks** dengan **tier tinggi**
   - **Kelebihan**

     Lebih efisien dengan mengurangi ruang pencarian (waktu pencarian lebih cepat).
   - **Kekurangan**

     Implementasi lebih **kompleks** dan membutuhkan lebih banyak data untuk pencarian mundur.
   - **Cocok untuk**

     Elemen dengan **tier tinggi** (tier 6+), elemen yang sulit ditemukan dengan BFS atau DFS.

- **Pemilihan Mode**

  1) **Single Recipe Mode**
     - **Kapan digunakan**

       Ketika hanya membutuhkan **satu jalur optimal** untuk membuat elemen target.
     - **Kelebihan**

       Lebih **cepat**, hasil yang sederhana dan mudah dipahami
     - **Kekurangan**

       Tidak menunjukkan alternatif yang mungkin lebih efisien
     - **Cocok untuk**

       Pencarian **langsung dan cepat** pada elemen sederhana.

  2) **Multiple Recipe Mode**

- **Kapan digunakan**
  Untuk menemukan **beberapa jalur** berbeda untuk membuat elemen target.
- **Kelebihan**
  Menunjukkan **alternatif jalur** dan bisa membantu menemukan cara terbaik untuk membuat elemen target
- **Kekurangan**
  Memerlukan lebih banyak waktu dan dapat menghasilkan **hasil yang lebih kompleks**
- **Cocok untuk**
  Eksplorasi mendalam atau elemen dengan banyak kemungkinan jalur.

- **Rekomendasi Kombinasi**
  1) **Elemen Tier Rendah (0–3)**
     BFS + Single Recipe.
  2) **Elemen Tier Menengah (4–6**
     Bidirectional + Single Recipe.
  3) **Elemen Kompleks (7+)**
     Bidirectional + Multiple Recipe.
  4) **Eksplorasi Alternatif**
     DFS + Multiple Recipe.

5. **Interpretasi Hasil Visualisasi**

   Hasil pencarian dalam aplikasi ini divisualisasikan dalam dua format utama: **Recipe Path** dan **Tree Visualization**. Berikut adalah cara untuk menginterpretasikan kedua format visualisasi:

   - **Recipe Path Interpretation**
     1) **Format langkah**
        Setiap langkah dalam pencarian disusun dalam format [Ingredient 1] + [Ingredient 2] → [Result].
     2) **Urutan langkah**
        Langkah-langkah disusun dari elemen **target** (atas) hingga elemen **dasar** (bawah), menunjukkan proses pembuatan secara berurutan.

3) **Informasi tier**

   Setiap elemen dalam langkah menunjukkan **tier** elemen tersebut, memberikan gambaran tentang tingkat kompleksitas dalam proses pembuatan.

- **Tree Visualization Interpretation**

1) **Node warna hijau**

   Menandakan **elemen target**.

2) **Node warna biru**

   Menandakan **elemen dasar** (misalnya: Fire, Earth, Water, Air).

3) **Node warna abu-abu**

   Menandakan **elemen perantara**.

4) **Edge/garis**

   Menunjukkan kombinasi dua elemen yang membentuk elemen berikutnya. Arah garis mengindikasikan arah dari **ingredient** ke **result**.

5) **Struktur vertikal**

   Elemen disusun berdasarkan tier, dengan elemen dasar di bawah dan elemen target di atas.

6) **Cabang**

   Menunjukkan **jalur alternatif** atau jika elemen berbagi ingredient.

- **Statistik Pencarian**

1) **Nodes visited**

   Jumlah **node** (elemen) yang telah dieksplorasi selama pencarian.

2) **Execution time**

   Waktu yang diperlukan untuk menemukan jalur, berguna untuk mengevaluasi efisiensi algoritma.

3) **Path length**

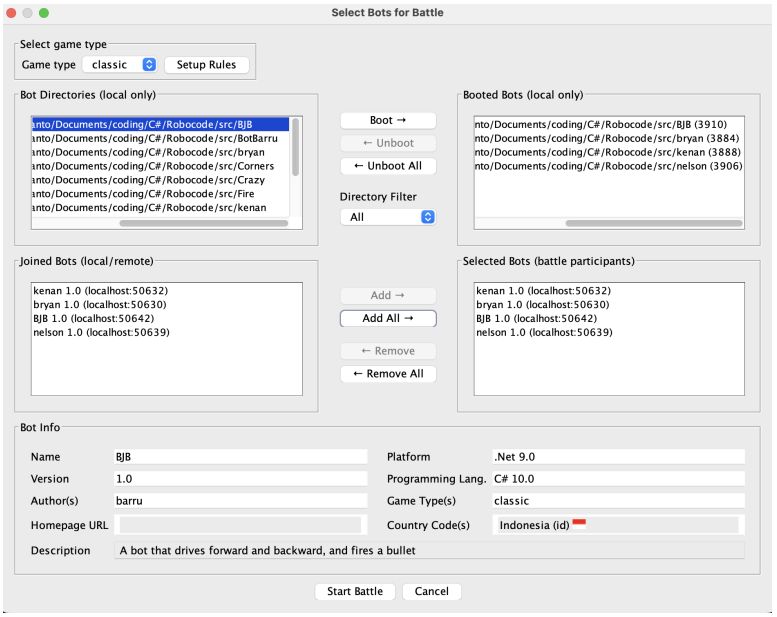
   Jumlah **langkah** dalam recipe, yang mencerminkan **kompleksitas** pembuatan elemen.

Pemahaman yang baik mengenai interpretasi visualisasi membantu pengguna untuk lebih mudah memahami jalur kombinasi elemen dan bagaimana proses pencarian dilakukan dalam aplikasi.

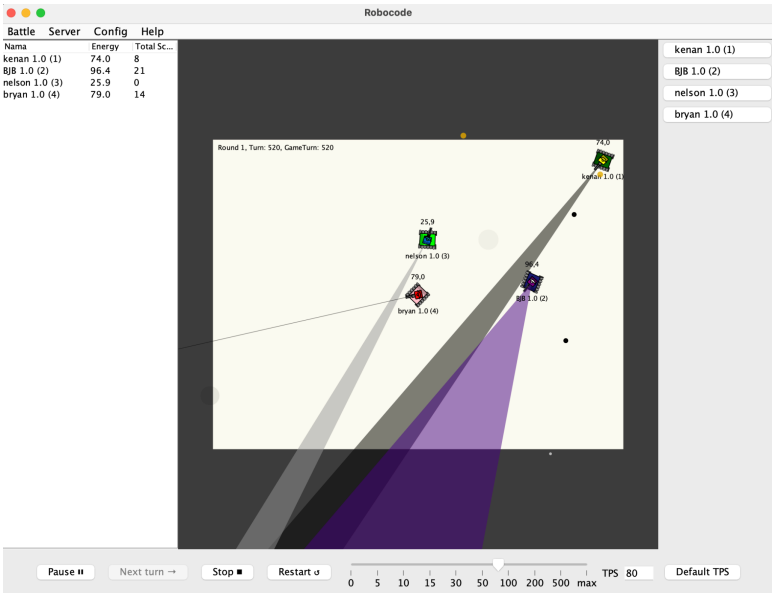
4.3 Hasil Pengujian

4.3.1 Pengujian Elemen Sederhana

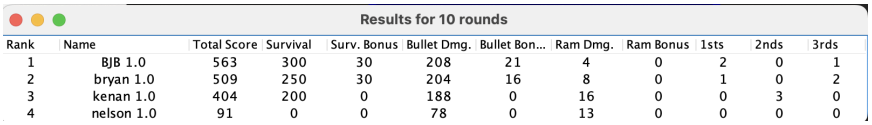
1. Booting



2. Playing



3. Leaderboard

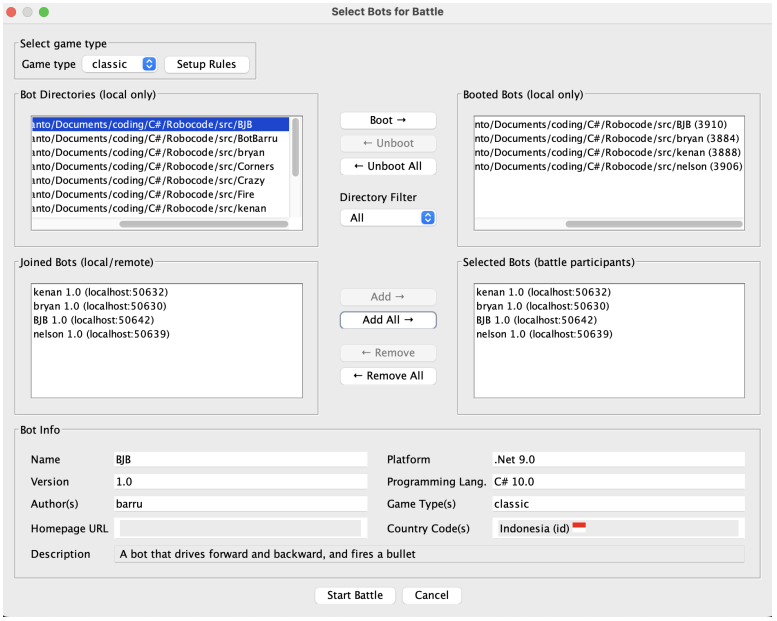


Results for 10 rounds

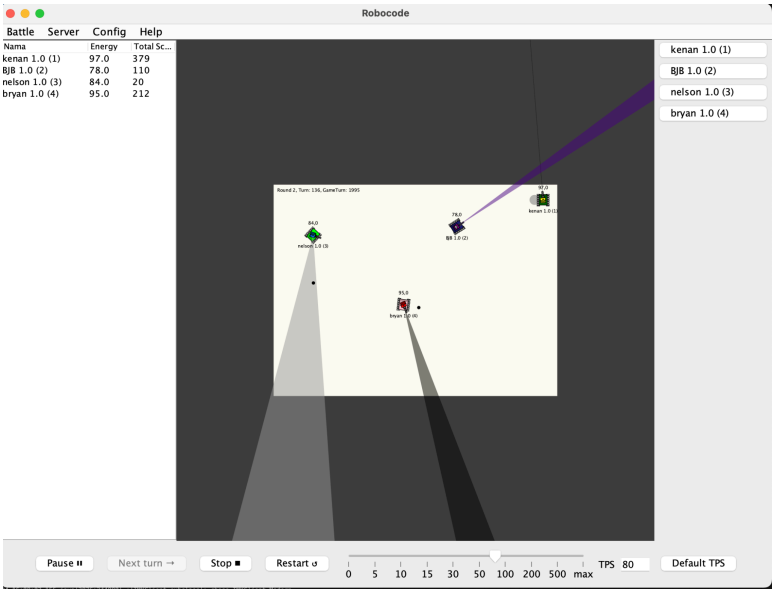
Rank	Name	Total Score	Survival	Surv. Bonus	Bullet Dmg.	Bullet Bon...	Ram Dmg.	Ram Bonus	1sts	2nds	3rds
1	BJB 1.0	563	300	30	208	21	4	0	2	0	1
2	bryan 1.0	509	250	30	204	16	8	0	1	0	2
3	kenan 1.0	404	200	0	188	0	16	0	0	3	0
4	nelson 1.0	91	0	0	78	0	13	0	0	0	0

4.3.2 Pengujian Elemen Kompleks

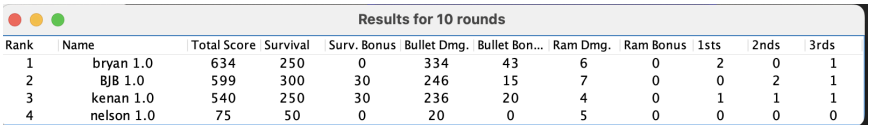
1. Booting



2. Playing



3. Leaderboard

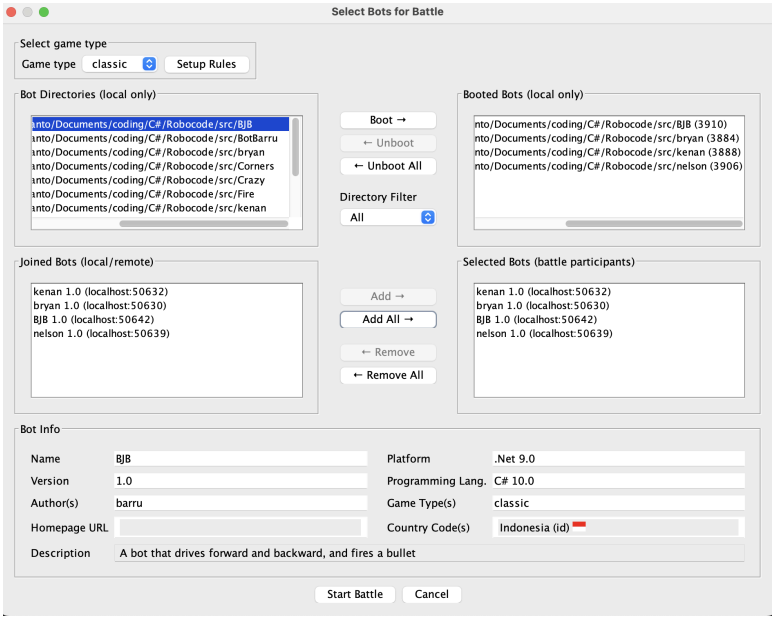


Results for 10 rounds

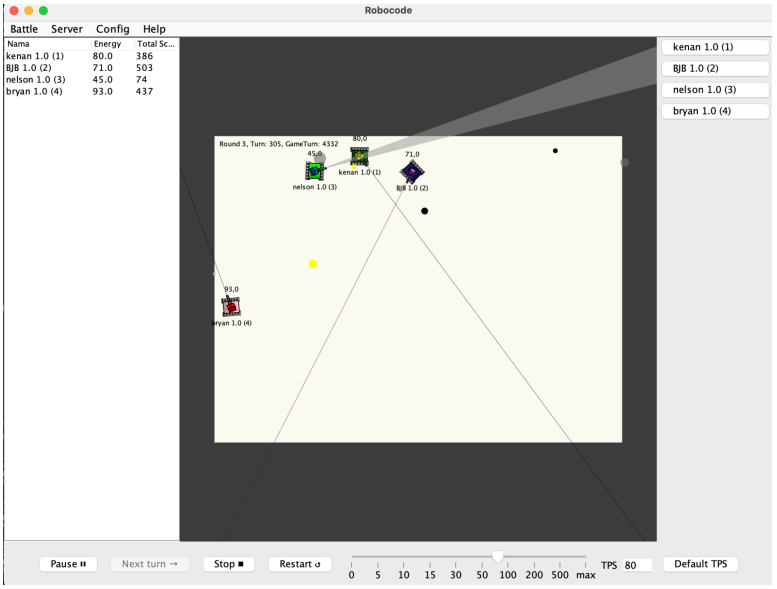
Rank	Name	Total Score	Survival	Surv. Bonus	Bullet Dmg.	Bullet Bon...	Ram Dmg.	Ram Bonus	1sts	2nds	3rds
1	bryan 1.0	634	250	0	334	43	6	0	2	0	1
2	BJB 1.0	599	300	30	246	15	7	0	0	2	1
3	kenan 1.0	540	250	30	236	20	4	0	1	1	1
4	nelson 1.0	75	50	0	20	0	5	0	0	0	0

4.3.3 Pengujian Multiple Recipe

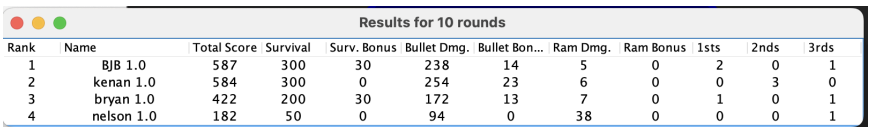
1. Booting



2. Playing



3. Leaderboard



Results for 10 rounds

Rank	Name	Total Score	Survival	Surv. Bonus	Bullet Dmg.	Bullet Bon...	Ram Dmg.	Ram Bonus	1sts	2nds	3rds
1	BJB 1.0	587	300	30	238	14	5	0	2	0	1
2	kenan 1.0	584	300	0	254	23	6	0	0	3	0
3	bryan 1.0	422	200	30	172	13	7	0	1	0	1
4	nelson 1.0	182	50	0	94	0	38	0	0	0	1

Bab V
Penutup

5.1 Kesimpulan

w

5.2 Saran

w

5.3 Refleksi

w

Lampiran

Link Github Repository : https://s.hmif.dev/GithubTrioKwekKwek

Link Video Youtube : https://s.hmif.dev/YoutubeTrioKwekKwek

Link Website : https://s.hmif.dev/WebsiteTrioKwekKwek

Link Laporan : https://s.hmif.dev/LaporanTrioKwekKwek

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

Daftar Pustaka

Little Alchemy 2. (n.d.). Little Alchemy 2. Retrieved from https://littlealchemy2.com/

Little-Alchemy Fandom. (n.d.). All elements in Little Alchemy 2. Retrieved from https://little-alchemy.fandom.com/wiki/Elements_(Little_Alchemy_2)

Mozilla Developer Network. (n.d.). Client-Server architecture. Retrieved from https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/Client-Server_overview

Amazon Web Services. (n.d.). What is an API? Retrieved from https://aws.amazon.com/what-is/api/

Next.js. (n.d.). Next.js Documentation. Retrieved from https://nextjs.org/docs

Go Documentation. (n.d.). Go Documentation. Retrieved from https://go.dev/doc/

Gin Gonic. (n.d.). Gin: The web framework written in Go. Retrieved from https://gin-gonic.com/

PuerkitoBio. (n.d.). goquery: A Go library for HTML scraping. Retrieved from https://github.com/PuerkitoBio/goquery

Go Documentation. (n.d.). Effective Go Concurrency. Retrieved from https://go.dev/doc/effective_go#concurrency