

Laporan Tugas Besar II Strategi Algoritma
Pemanfaatan Algoritma BFS dan DFS dalam Pencarian Recipe pada
Permainan Little Alchemy 2
RecipePlayground



Adiel Rum
10123004
10123004@mahasiswa.itb.ac.id

Muhammad Fathur Rizky
13523105
13523105@std.stei.itb.ac.id

Ahmad Wafi Idzharulhaqq
13523131
13523131@std.stei.itb.ac.id

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2025

Daftar Isi

BAB I: Deskripsi Tugas	2
BAB II: Landasan Teori	3
DFS (Depth-First Search)	3
BFS (Breadth-First Search)	3
Bidirectional Search	3
BAB III: Analisis Pemecahan Masalah	4
Analisis Permasalahan	4
Langkah-Langkah Pemecahan Masalah	4
Proses Pemetaan Masalah Menjadi Elemen-Elemen Algoritma DFS dan BFS	4
Fitur Fungsional dan Arsitektur Aplikasi Web yang Dibangun	4
Pemilihan Strategi Pencarian	4
Pengaturan Resep Maksimum	5
Visualisasi Pohon Resep	5
Perpindahan Dinamis Antar Strategi	5
Contoh Ilustrasi Kasus	5
BAB IV: Implementasi dan Pengujian	6
Spesifikasi Teknis Program	6
Struktur Data	6
The Scraper Algorithm	6
The DFS Algorithm (Mutex for Multithreading)	8
The BFS Algorithm (Mutex for Multithreading)	10
Penggunaan Program	14
Interface dan Alur Penggunaan Program	14
Fitur-Fitur Utama	15
Fitur Tambahan	15
Hasil Pengujian	15
Analisis Hasil Pengujian	18
BAB V: Kesimpulan dan Saran	19
5.1 Kesimpulan	19
5.2 Saran	20
Lampiran	20
Daftar Pustaka	21

BAB I: Deskripsi Tugas

Little Alchemy 2 adalah sebuah permainan berbasis web dan aplikasi yang dikembangkan oleh Recloak dan dirilis pada tahun 2017. Permainan ini merupakan sekuel dari Little Alchemy 1 yang dirilis pada tahun 2010. Tujuan utama dari Little Alchemy 2 adalah untuk menciptakan 720 elemen berbeda dengan memulai dari empat elemen dasar: air, api, tanah, dan udara.

Mekanisme permainan melibatkan pemain yang menggabungkan dua elemen dengan cara drag and drop. Jika kombinasi kedua elemen tersebut valid, maka akan tercipta elemen baru. Namun, jika kombinasi tidak valid, tidak akan terjadi apa-apa. Elemen turunan yang berhasil dibuat dapat digunakan kembali untuk membentuk elemen-elemen lainnya. Permainan ini dapat diakses melalui peramban web, serta pada perangkat Android dan iOS.

Dalam konteks Tugas Besar pertama Strategi Algoritma, mahasiswa diminta untuk menyelesaikan permainan Little Alchemy 2 dengan menerapkan strategi Depth First Search (DFS) dan Breadth First Search (BFS).

Komponen-komponen utama dalam permainan Little Alchemy 2 adalah sebagai berikut:

- **Elemen Dasar:** Terdapat empat elemen dasar yang menjadi titik awal permainan, yaitu:
 - Air (water)
 - Api (fire)
 - Tanah (earth)
 - Udara (air) Keempat elemen dasar ini nantinya akan dikombinasikan untuk menghasilkan 720 elemen turunan.



Gambar 1. Elemen dasar pada Little Alchemy 2

- **Elemen Turunan:** Terdapat total 720 elemen turunan yang dapat diciptakan. Elemen-elemen ini dikategorikan ke dalam beberapa tingkatan (tier) berdasarkan tingkat kesulitan dan jumlah langkah yang dibutuhkan untuk menciptakannya. Setiap elemen

turunan memiliki "resep" yang terdiri dari kombinasi elemen lain atau bahkan elemen itu sendiri.

- **Mekanisme Kombinasi (Combine Mechanism):** Untuk memperoleh elemen turunan, pemain harus menggabungkan dua elemen. Elemen turunan yang sudah berhasil didapatkan dapat digunakan kembali oleh pemain sebagai bahan untuk membentuk elemen-elemen lainnya.

BAB II: Landasan Teori

DFS (Depth-First Search)

Depth-First Search (DFS) adalah sebuah algoritma penelusuran graf atau pohon yang menjelajahi sejauh mungkin sepanjang setiap cabang sebelum melakukan backtracking. Algoritma ini dimulai dari simpul akar (atau simpul acak dalam kasus graf) dan menjelajahi sedalam mungkin di sepanjang setiap cabang sebelum mundur. Artinya, DFS akan mengunjungi semua anak dari sebuah simpul sebelum mengunjungi saudara-saudaranya. Implementasi DFS umumnya menggunakan struktur data stack (tumpukan), baik secara implisit melalui rekursi atau secara eksplisit. Keuntungan utama DFS adalah kebutuhan memorinya yang relatif kecil jika dibandingkan dengan BFS pada graf yang lebar, namun DFS tidak menjamin penemuan solusi terpendek dan dapat terjebak dalam cabang tak hingga jika tidak diimplementasikan dengan hati-hati pada graf siklik. Menurut Cormen et al. (2009), DFS sangat berguna untuk berbagai aplikasi seperti topological sorting, menemukan komponen terhubung kuat, dan mendeteksi siklus dalam graf.

BFS (Breadth-First Search)

Breadth-First Search (BFS) adalah algoritma penelusuran graf atau pohon yang menjelajahi semua simpul pada kedalaman saat ini sebelum pindah ke simpul pada kedalaman berikutnya. Algoritma ini dimulai dari simpul akar dan mengunjungi semua tetangga langsung dari simpul akar, kemudian semua tetangga yang belum dikunjungi dari tetangga-tetangga tersebut, dan seterusnya, lapis demi lapis. BFS biasanya diimplementasikan menggunakan struktur data queue (antrean) untuk menyimpan simpul-simpul yang akan dikunjungi. Salah satu karakteristik penting BFS adalah kemampuannya untuk menemukan jalur terpendek antara dua simpul dalam graf tak berbobot, diukur berdasarkan jumlah sisi. Seperti yang dijelaskan oleh Russell & Norvig (2020), BFS bersifat lengkap, artinya akan selalu menemukan solusi jika ada, dan optimal jika biaya langkahnya seragam.

Bidirectional Search

Bidirectional Search adalah strategi pencarian yang melakukan dua pencarian secara simultan: satu pencarian maju dari keadaan awal dan satu pencarian mundur dari keadaan tujuan. Pencarian ini akan berhenti ketika kedua batas pencarian bertemu di tengah. Ide utama di balik bidirectional search adalah bahwa melakukan dua pencarian pada kedalaman $d/2$ (dengan asumsi faktor percabangan b) secara signifikan lebih efisien daripada satu pencarian pada kedalaman d . Secara teoritis, kompleksitasnya dapat berkurang dari $O(b^d)$ menjadi $O(2b^{d/2})$ atau $O(b^{d/2})$. Russell & Norvig (2020) menjelaskan bahwa efektivitas bidirectional search sangat bergantung pada kemampuan untuk melakukan pencarian mundur secara efisien dan

mendeteksi kapan kedua pencarian bertemu, yang mungkin tidak selalu mudah atau mungkin untuk semua jenis masalah. Implementasinya bisa menjadi lebih kompleks karena memerlukan pengelolaan dua proses pencarian dan pemeriksaan pertemuan.

BAB III: Analisis Pemecahan Masalah

Analisis Permasalahan

Permasalahan yang diangkat adalah bagaimana menelusuri langkah demi langkah pembuatan elemen dalam Little Alchemy 2. Kita membutuhkan cara untuk merepresentasikan elemen beserta resepnya sebagai struktur yang mudah ditelusuri, serta mekanisme untuk mencegah eksplorasi tak terbatas akibat siklus di dalam resep.

Langkah-Langkah Pemecahan Masalah

- Muat Data & Inisialisasi: Ambil daftar elemen dan resep dari sumber (JSON hasil scraping) ke memori.
- Terima Parameter: Pengguna menentukan target elemen, strategi pencarian (DFS, BFS, atau Bidirectional), dan jumlah resep yang ingin ditemukan.
- Jalankan Algoritma: Panggil modul solver yang melakukan traversal graf dengan memperhatikan max recipes.
- Bentuk Output: Kemas hasil traversal sebagai nested map (pohon resep) yang nantinya siap divisualisasikan.
- Bentuk Visualisasi: Visualisasikan hasil berdasarkan metode yang pengguna pilih.

Proses Pemetaan Masalah Menjadi Elemen-Elemen Algoritma DFS dan BFS

- Struktur Graf: Setiap elemen adalah simpul (node); setiap resep dua bahan menjadi anak (child) dari node tersebut. Misal $A \rightarrow [B, C]$ bila A dibuat dari B+C.
- DFS (Depth-First Search): Menyelami satu jalur anak hingga simpul daun atau maxRecipes, kemudian backtrack untuk mencari jalur lain—bagus untuk eksplorasi mendalam, tapi tidak menjamin jalur terpendek.
- BFS (Breadth-First Search): Menjelajah per tingkat; semua anak pada kedalaman n diproses sebelum lanjut ke $n+1$ —menjamin menemukan jalur dengan jumlah langkah minimal.
- Bidirectional Search: Dua BFS berjalan simultan—satu mundur dari target, satu maju dari elemen dasar—dan berhenti ketika front bertemu, mempercepat pencarian di graf besar.

Fitur Fungsional dan Arsitektur Aplikasi Web yang Dibangun

Pemilihan Strategi Pencarian

Pengguna dapat memilih satu dari tiga algoritma (DFS, BFS, Bidirectional) untuk menelusuri jalur pembuatan elemen. Ini memungkinkan mereka membandingkan karakteristik tiap strategi—misalnya, jalur terpendek (BFS) vs. eksplorasi mendalam (DFS) vs. pencarian dua arah (Bidirectional).

Pengaturan Resep Maksimum

Untuk mengontrol kompleksitas hasil dan mencegah siklus tak berujung, tersedia opsi memasukkan batas banyak resep yang ingin ditemukan. Dengan demikian, pengguna bisa menyesuaikan seberapa jauh pohon resep dieksplorasi.

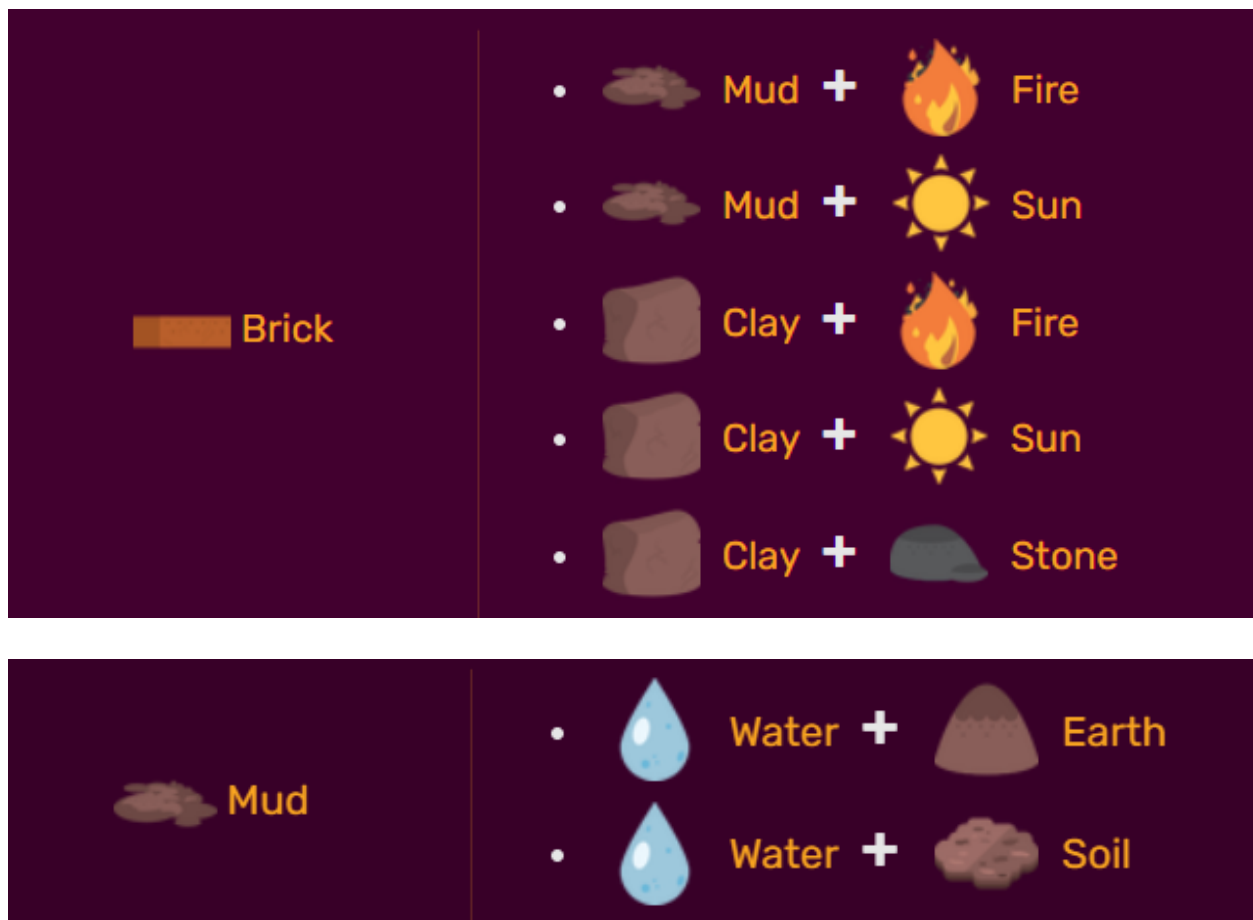
Visualisasi Pohon Resep

Hasil pencarian disajikan sebagai struktur hirarkis yang interaktif: simpul dapat diperluas/ditutup, menampilkan anak-anaknya, dan memberi tooltip nama elemen saat diarahkan kursor. Ini membantu memahami alur pembuatan secara intuitif.

Perpindahan Dinamis Antar Strategi

Tanpa memuat ulang halaman, pengguna bisa berganti strategi pencarian dan melihat perubahan pohon resep secara real-time sehingga langsung merasakan perbedaan DFS, BFS, dan Bidirectional.

Contoh Ilustrasi Kasus



Misalnya targetnya Brick. Dengan BFS dan maxRecipes=2, algoritma pertama menelusuri resep Brick(misal “Mud”+“Fire”), kemudian menelusuri resep Mud dan Fire di kedalaman berikutnya, sehingga ditemukan jalur terpendek dalam tiga langkah. Jika menggunakan DFS, yang tampil adalah satu jalur penuh (Brick → Mud → Water) sebelum menjajal cabang lain.

Dengan menggunakan BFS, maka jalur yang dikunjungi adalah resep terdekatnya terlebih dahulu, dalam hal ini adalah (Brick → Mud → Fire).

BAB IV: Implementasi dan Pengujian

Spesifikasi Teknis Program

Struktur Data

```
type Data struct {
    Elements []Element `json:"elements"`
}

type Element struct {
    Name      string    `json:"name"`
    Recipes   [][]string `json:"recipes"`
    Tier      int       `json:"tier"`
}
```

Terdapat dua struktur data utama yang digunakan pada pemodelan, yakni Data dan Element. Data digunakan untuk menyimpan data yang di-*load* dari JSON dan disimpan dalam bentuk slice of Element. Setiap objek Element merepresentasikan suatu elemen dalam sistem, yang memiliki tiga atribut utama: Name, Recipes, dan Tier. Atribut Name berupa string yang menunjukkan nama elemen tersebut. Atribut Recipes merupakan slice dua dimensi bertipe string, yang menyimpan kombinasi resep (pasangan elemen lain) yang dapat digunakan untuk membuat elemen tersebut. Sementara itu, atribut Tier berupa integer yang menunjukkan tingkatan atau level dari elemen tersebut dalam hierarki sistem. Struktur ini mempermudah proses pencarian dan komposisi elemen berdasarkan data yang diambil dari file JSON.

The Scraper Algorithm

```
func Scrape(filename string) error {
    const url =
        "https://little-alchemy.fandom.com/wiki/Elements_(Little_Alchemy_2)"

    res, err := http.Get(url)
    defer res.Body.Close()
    ...
    doc, err := goquery.NewDocumentFromReader(res.Body)

    var elements []model.Element
    tier := 0
    tables.Each(func(i int, tbl *goquery.Selection) {
        if i == 1 {
            return
        }

        tbl.Find("tr").Each(func(rowIdx int, row *goquery.Selection) {
            if rowIdx == 0 {
                return
            }
            tds := row.Find("td")
```

```

    if tds.Length() < 2 {
        return
    }

    var name string
    tds.Eq(0).Contents().EachWithBreak(func(_ int, node *goquery.Selection)
bool {
    if goquery.NodeName(node) == "a" {
        if title, ok := node.Attr("title"); ok {
            name = title
            return false
        }
    }
    return true
})
if name == "" {
    return
}

var recipes [][]string
tds.Eq(1).Find("ul > li").Each(func(_ int, li *goquery.Selection) {
    var combo []string
    li.Find("a").Each(func(_ int, a *goquery.Selection) {
        if t, ok := a.Attr("title"); ok {
            combo = append(combo, t)
        }
    })
    if len(combo) > 0 {
        recipes = append(recipes, combo)
    }
})

elements = append(elements, model.Element{
    Name:    name,
    Recipes: recipes,
    Tier:    tier,
})
})

tier++
})

...
}

```

Metode scraping yang digunakan dalam fungsi Scrape bertujuan untuk mengambil data elemen dari situs [Little Alchemy 2 Fandom](#). Proses dimulai dengan melakukan permintaan HTTP GET ke URL yang berisi daftar elemen permainan. Setelah memastikan bahwa status respons HTTP valid (kode 200), HTML yang diterima di-parse menggunakan pustaka goquery untuk memudahkan navigasi dan ekstraksi elemen DOM. Fungsi ini secara khusus mencari tabel-tabel HTML dengan kelas list-table col-list icon-hover, yang mengandung data elemen dan resepnya. Setiap baris tabel mewakili satu elemen, dengan kolom pertama berisi nama elemen dan kolom kedua berisi daftar kombinasi resep yang dapat digunakan untuk membuat elemen tersebut. Nama elemen diekstrak dari atribut title pada tag <a>, sementara resep diambil dari

struktur nested `<a>>`, dikumpulkan dalam slice dua dimensi. Tingkatan elemen ditentukan berdasarkan urutan tabel, kecuali tabel kedua yang diabaikan karena tidak relevan. Seluruh data elemen beserta resep dan tingkatannya kemudian disimpan dalam struktur `model.Data`, dan di-serialize menjadi file JSON dengan format yang rapi menggunakan `json.Encoder`.

The DFS Algorithm (Mutex for Multithreading)

```
import (
    "sync/atomic"
)
```

Di bagian atas diimpor paket `sync/atomic` yang digunakan untuk operasi atomik pada variabel integer, memastikan bahwa operasi tersebut aman untuk digunakan dalam lingkungan multithreading.

```
func Dfs(rootElementName string, maxRecipes int64) interface{} {
    if elementsMapGlobal == nil {
        return map[string]string{"error": "Element data not initialized"}
    }

    var totalCount int64
    atomic.StoreInt64(&visitedNodeCount, 0)
    return dfsChunked(rootElementName, &totalCount, maxRecipes)
}
```

Fungsi `Dfs` adalah entry point. Ia pertama-tama memeriksa apakah data elemen (`elementsMapGlobal`) telah diinisialisasi—jika belum, langsung return error. Lalu ia Mengatur `totalCount` dan mereset `visitedNodeCount` secara atomik untuk memastikan variabel berada dalam kondisi awal. Selanjutnya dialkuan pemanggilan secara rekursif terhadap `dfsChunked` menggunakan `totalCount` yang telah diinisiasi sebelumnya bersama `rootElementName` dan `maxRecipe`.

```
func dfsChunked(elementName string, totalCount *int64, maxRecipes int64)
interface{} {
    if atomic.LoadInt64(totalCount) >= maxRecipes {
        atomic.AddInt64(&visitedNodeCount, 1)
        return elementName
    }
    ...
}
```

Di awal `dfsChunked`, kode melakukan pemeriksaan apakah `totalCount` telah mencapai jumlah `maxRecipes`. Jika ya, dilakukan penambahan terhadap `visitedNodeCount` dan kode mengembalikan `elementName`.

```
eData, exists := elementsMapGlobal[elementName]

if !exists || len(eData.Recipes) == 0 || eData.Tier == 0 {
    atomic.AddInt64(&visitedNodeCount, 1)
    return elementName
}
```

```

}

atomic.AddInt64(&visitedNodeCount, 1)

```

Berikutnya, kode memverifikasi keabsahan elemen: jika elementName tidak ditemukan di elementsMapGlobal, tidak memiliki resep, atau berada di tier dasar (Tier == 0), maka dianggap daun. Dilakukan penambahan visitedNodeCount dan elementName dikembalikan ke luar fungsi. Sebaliknya, dilakukan penambahan visitedNodeCount dan fungsi masih melakukan pemrosesan.

```

currentTier := eData.Tier
recipes := make([][]string, 0, len(eData.Recipes))

for _, rec := range eData.Recipes {
    if len(rec) != 2 {
        continue
    }
    c1, c2 := rec[0], rec[1]
    child1, ok1 := elementsMapGlobal[c1]
    child2, ok2 := elementsMapGlobal[c2]

    if !ok1 || !ok2 || child1.Tier >= currentTier || child2.Tier >=
currentTier {
        continue
    }
    recipes = append(recipes, []string{c1, c2})
}

```

Di tahap ini, kode memfilter resep biner (pasangan dua elemen) yang valid berdasarkan tier. Dari setiap rec, ia memastikan kedua anak (c1, c2) benar-benar ada, dan tier-nya lebih rendah daripada currentTier. Hanya resep yang memenuhi kriteria ini yang dimasukkan ke slice recipes.

```

var results [][]interface{}

for _, rec := range recipes {

    if atomic.LoadInt64(totalCount) >= maxRecipes {
        break
    }

    left := dfsChunked(rec[0], totalCount, maxRecipes)
    right := dfsChunked(rec[1], totalCount, maxRecipes)

    leftData, leftOk := elementsMapGlobal[rec[0]]
    rightData, rightOk := elementsMapGlobal[rec[1]]
    if leftOk && rightOk && leftData.Tier == 0 && rightData.Tier == 0 {
        atomic.AddInt64(totalCount, 1)
    }

    results = append(results, []interface{}{left, right})
}

```

```
return map[string]interface{}{elementName: results}
```

Selanjutnya berjalan loop depth-first pada setiap resep yang lolos filter. Di setiap iterasi, kode kembali memeriksa counter (apakah sudah mencapai maxRecipes) dan menaikkannya secara atomik. Jika belum, ia merekursi ke dfsChunked untuk anak kiri dan kanan. Jika salah satu cabang menghasilkan nil, pasangan diabaikan, sehingga tidak ada null di output. Pasangan valid ditambahkan ke results.

```
func GetVisitedNodeCount() int64 {  
    return atomic.LoadInt64(&visitedNodeCount)  
}
```

Fungsi ini digunakan untuk mengembalikan jumlah node yang telah dikunjungi secara atomik.

The BFS Algorithm (Mutex for Multithreading)

```
import (  
    "runtime"  
    "sync"  
    "sync/atomic"  
)  
  
var (  
    visitedNodeCount int64  
)  
  
type node struct {  
    Name      string  
    Children [][]*node  
}  
  
type bfsEvent struct {  
    parent *node  
    child1 *node  
    child2 *node  
}
```

Di bagian atas diimpor runtime, sync, dan sync/atomic untuk mendukung multithreading dan operasi atomik. Setelah itu, dilakukan deklarasi variabel global visitedNodeCount yang digunakan untuk menghitung jumlah node dikunjungi, struktur data node untuk merepresentasikan elemen dalam pencarian, dan bfsEvent yang menyimpan informasi hubungan antar node.

```
func Bfs(rootElementName string, maxRecipes int64) interface{} {  
  
    if elementsMapGlobal == nil {  
        return map[string]string{"error": "Element data not initialized"}  
    }  
  
    root := &node{Name: rootElementName}  
    currentLevel := []*node{root}  
    workers := runtime.NumCPU()
```

```
var totalCount int64

atomic.StoreInt64(&visitedNodeCount, 0)
```

Fungsi Bfs adalah titik masuk. Pertama ia mengecek apakah data elementsMapGlobal sudah diisi—jika belum, langsung kembalikan error. Jika sudah, dilanjutkan dengan inisialisasi root, mengatur current level untuk memulai BFS, dan variabel workers berdasarkan jumlah CPU yang nantinya menentukan jumlah goroutine yang a. Variabel totalCount dideklarasikan, dan dilakukan reset terhadap visitedNodeCount secara atomik.

```
for len(currentLevel) > 0 && atomic.LoadInt64(&totalCount) < maxRecipes {

    eventsPerWorker := make([][]bfsEvent, workers)

    var wg sync.WaitGroup
    wg.Add(workers)

    n := len(currentLevel)

    chunkSize := (n + workers - 1) / workers
```

Loop utama berjalan selama masih ada simpul di currentLevel dan totalCount belum mencapai maxRecipes. Di setiap iterasi, dibuat slice eventsPerWorker untuk menampung bfsEvent tiap goroutine. WaitGroup di-setup untuk menunggu seluruh worker selesai. Data currentLevel kemudian dibagi ke dalam workers chunk berdasarkan ukuran chunkSize.

```
for w := 0; w < workers; w++ {
    start := w * chunkSize
    end := start + chunkSize

    if start >= n {
        wg.Done()
        continue
    }

    if end > n {
        end = n
    }
    chunk := currentLevel[start:end]

    go func(id int, parents []*node) {
        defer wg.Done()
        localEvents := eventsPerWorker[id]

        for _, parent := range parents {

            if atomic.LoadInt64(&totalCount) >= maxRecipes {
                break
            }

            atomic.AddInt64(&visitedNodeCount, 1)

            eData, exists := elementsMapGlobal[parent.Name]
```

```

        if !exists || len(eData.Recipes) == 0 || eData.Tier == 0
    {

        continue
    }

    parentTier := eData.Tier

    for _, rec := range eData.Recipes {

        if atomic.LoadInt64(&totalCount) >= maxRecipes {
            break
        }

        if len(rec) != 2 {
            continue
        }
        c1Name, c2Name := rec[0], rec[1]

        child1Data, ok1 := elementsMapGlobal[c1Name]
        child2Data, ok2 := elementsMapGlobal[c2Name]

        if !ok1 || !ok2 || child1Data.Tier >= parentTier ||
child2Data.Tier >= parentTier {
            continue
        }

        child1 := &node{Name: c1Name}
        child2 := &node{Name: c2Name}

        localEvents = append(localEvents, bfsEvent{parent:
parent, child1: child1, child2: child2})

        if child1Data.Tier == 0 && child2Data.Tier == 0 {
            atomic.AddInt64(&totalCount, 1)
        }

        if atomic.LoadInt64(&totalCount) >= maxRecipes {
            break
        }
    }
    eventsPerWorker[id] = localEvents
}
}(w, chunk)
}
wg.Wait()

```

Di setiap goroutine, untuk setiap parent dalam chunk, pertama-tama dicek apakah totalCount telah mencapai batas maksimum. Jika belum, data elemen (eData) diambil dari cache untuk mendapatkan resep biner. Resep yang valid disaring berdasarkan tier, memastikan bahwa anak-anak memiliki tier lebih rendah dari induknya. Untuk setiap pasangan resep yang valid, totalCount ditambah secara atomik, dan simpul anak dibuat. Pasangan anak yang valid disimpan dalam buffer lokal localEvents. Setelah semua parent dalam chunk diproses, hasilnya

disimpan ke `eventsPerWorker`. Goroutine menandai tugasnya selesai dengan `defer wg.Done()`, dan `wg.Wait()` digunakan untuk menunggu semua goroutine selesai sebelum melanjutkan ke langkah berikutnya.

```
nextLevel := make([]*node, 0)
events := []bfsEvent{}
for _, buf := range eventsPerWorker {
    events = append(events, buf...)
}
for _, ev := range events {
    ev.parent.Children = append(ev.parent.Children,
[]*node{ev.child1, ev.child2})
    nextLevel = append(nextLevel, ev.child1, ev.child2)
}

currentLevel = nextLevel
}

return convertNode(root)
}
```

Setelah semua worker selesai, buffer `eventsPerWorker` diflatten menjadi satu slice `events`. Tiap `bfsEvent` dipakai untuk: (1) menempelkan anak ke `parent.Children`, dan (2) menambahkan `child1` dan `child2` ke `nextLevel`. Kemudian `currentLevel` di-update ke `nextLevel`, memulai iterasi BFS berikutnya. Setelah loop berakhir (karena level habis atau `maxRecipes` tercapai), pohon `root` dikonversi ke struktur siap-JSON lewat `convertNode`.

```
func convertNode(n *node) interface{} {
    if len(n.Children) == 0 {
        return n.Name
    }
    pairs := make([]interface{}, len(n.Children))
    for i, children := range n.Children {
        p1 := convertNode(children[0])
        p2 := convertNode(children[1])
        pairs[i] = []interface{}{p1, p2}
    }
    return map[string]interface{}{n.Name: pairs}
}
```

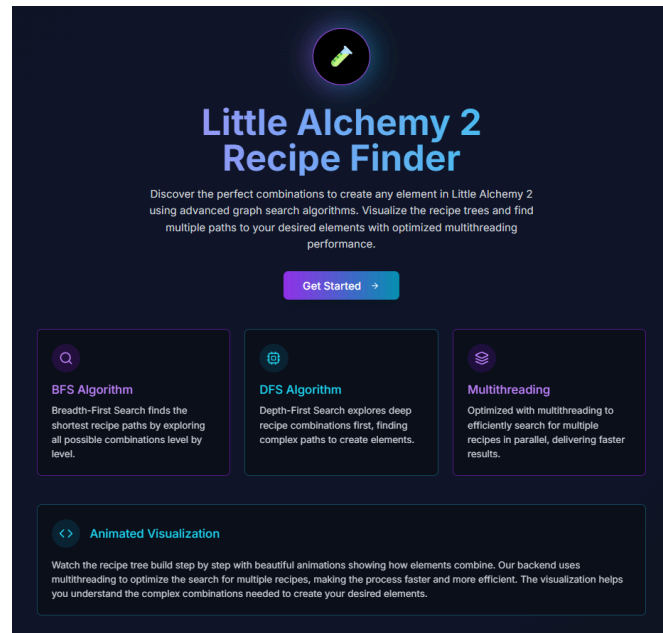
Kita total elemen ada berapakah bang & fitur mau kasih foto lgi atau tulisan aja (dah lengkap juga foto di interfacenya)

Fungsi `convertNode` menerjemahkan pohon node menjadi nested map/slice:

- Jika daun (`Children` kosong), kembalikan nama elemen sebagai string.
- Jika cabang, rekursi ke setiap anak, lalu kumpulkan pasangan `[left, right]` dalam slice `pairs`.
- Akhirnya bungkus `pairs` ke map dengan kunci `n.Name`. Hasilnya siap di-serialize menjadi JSON tanpa entri null.

Penggunaan Program

Interface dan Alur Penggunaan Program



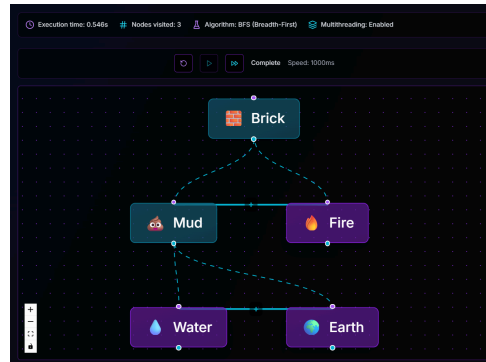
Gambar 4.1. Menu Utama

Setelah membuka aplikasi, pengguna akan disambut dengan interface yang terstruktur dengan baik. Terdapat judul dan keterangan fitur yang dimiliki aplikasi serta akses ke source code. Selanjutnya, pengguna dapat menekan tombol “Get Started” atau melakukan scroll ke bawah halaman utama untuk memasuki menu pencarian. Di dalam menu pencarian terdapat berbagai opsi yang dapat pengguna pilih untuk menemukan resep. Setelah itu, akan ditampilkan hasil berisi resep dari elemen yang dicari disertai keterangan waktu, banyak node, metode yang digunakan, dan keterangan multithreading. .



Gambar 4.2. Menu Pencarian Resep

Pada bagian Search Parameters, pengguna dapat memasukkan 3 input, yakni algoritma yang digunakan, elemen yang ingin dibentuk (target), dan banyak resep dari target ke elemen dasar (tier 0). Setelah itu, pengguna dapat menekan submit button (Find Recipes) untuk menemukan jalur resep dalam membuat elemen target.



Gambar 4.3. Tampilan Hasil

Setelah menekan Find Recipes, pengguna dapat melihat jalur dari target ke elemen dasar yang dapat membantu dalam melakukan pembuatan elemen target.

Fitur-Fitur Utama

- Pencarian Resep dengan Algoritma BFS: Pencarian resep dengan pendekatan Breadth-First Search untuk mendapatkan hasil pencarian lebih luas dan menyeluruh.
- Pencarian Resep dengan Algoritma DFS: Mendukung pencarian resep menggunakan Depth-First Search yang cocok untuk eksplorasi mendalam pada jalur resep tertentu.
- Multi-Recipes Finder: Fitur ini memungkinkan pengguna menentukan berapa banyak recipe yang akan ditampilkan dalam satu kali pencarian
- Pohon Resep Interaktif: Visualisasi resep dalam bentuk pohon yang dapat di-drag, di-zoom, dan disertai animasi sehingga memudahkan eksplorasi elemen resep secara visual dan dinamis.

Fitur Tambahan

Selain fitur untuk menemukan jalur, pengguna dapat melihat akses kode sumber program untuk keperluan pembelajaran atau pengembangan lebih lanjut.

Hasil Pengujian

Test Case #1 (No Element Selected)

Input/Output

🧪

Search Parameters

Algorithm

Breadth-First Search (BFS)

Target Element

Select element

Maximum Recipes

1

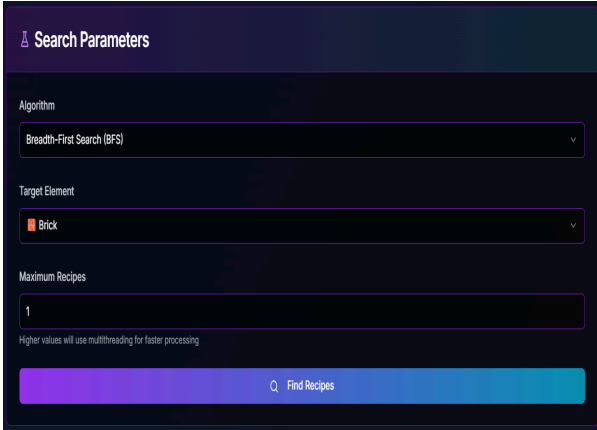
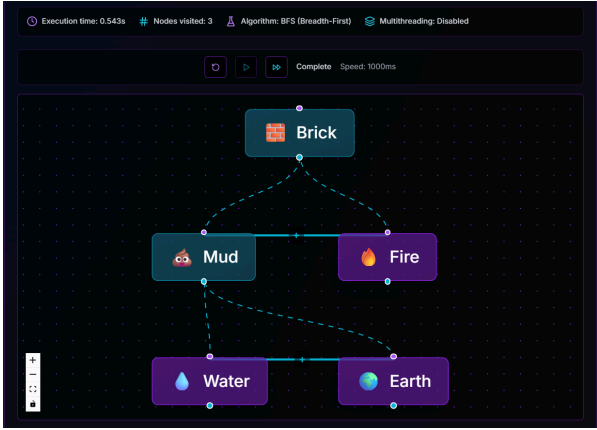
Higher values will use multithreading for faster processing

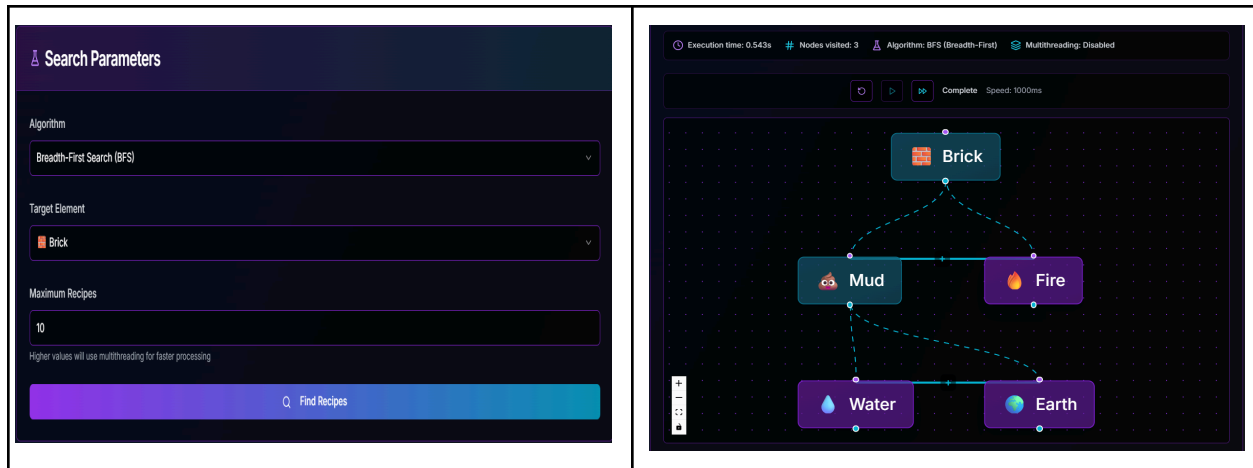
Find Recipes

Please select a target element

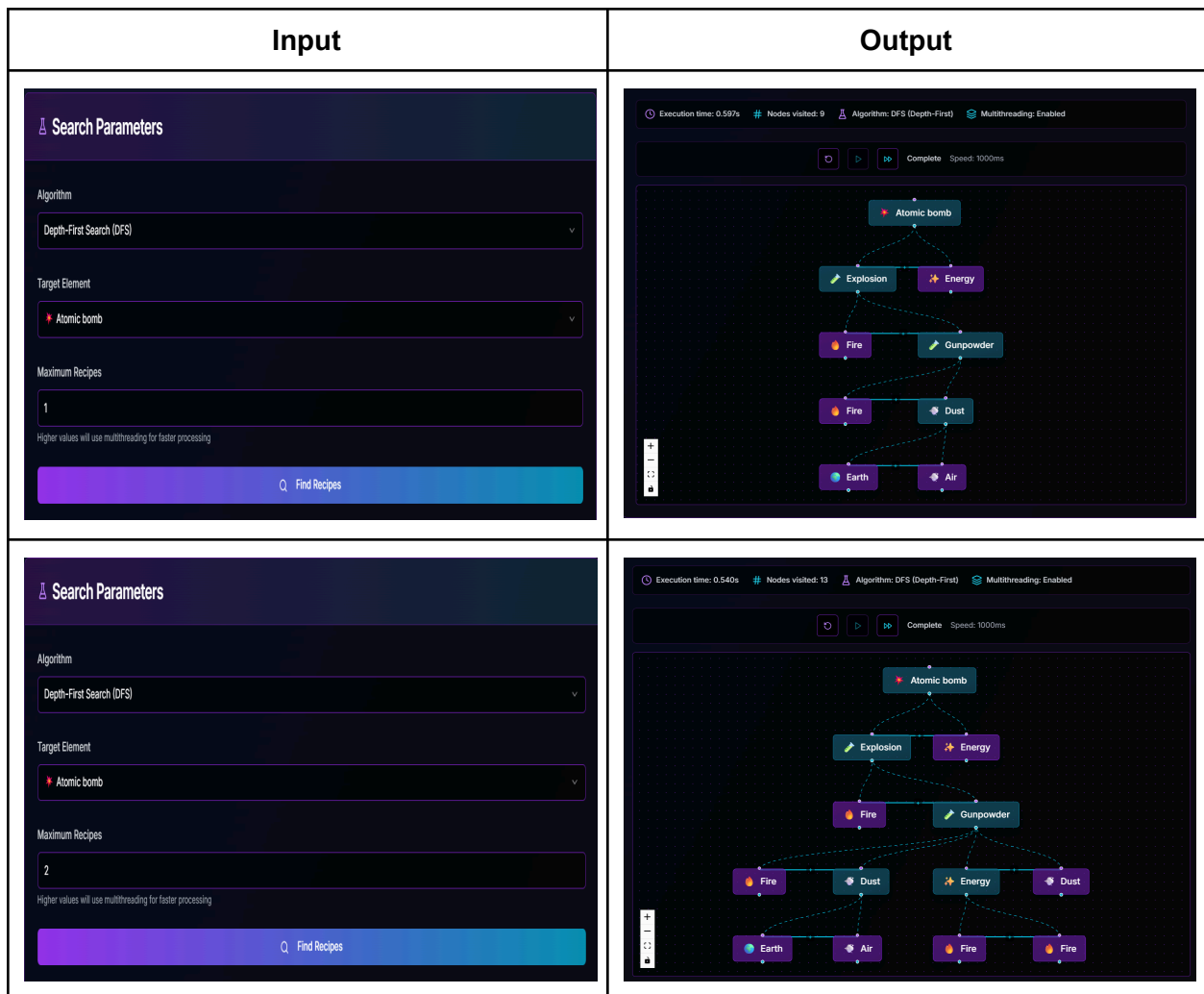
Test Case #2 (BFS: Positive Input)

Terlihat bahwa kedua output sama ketika dimasukkan input yang berbeda. Hal ini diakibatkan kesempitan pohon solusi dari Brick.






Input	Output
	







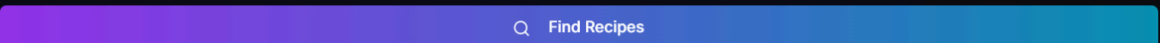
Test Case #3 (DFS: Positive Input)



Test Case #4 (Null Max Recipes)

Input/Output
<div> <div>  Search Parameters </div> <div> <div>Algorithm</div> <div>Breadth-First Search (BFS) </div> </div> <div> <div>Target Element</div> <div> Airplane </div> </div> <div> <div>Maximum Recipes</div> <div></div> </div> <div> <div>Higher values will use multithreading for faster processing</div> <div>  </div> </div> <div> <div>Please enter a valid number for maximum recipes</div> </div> </div>

Test Case #5 (Negative Max Recipes)

Input/Output
<div> <div>  Search Parameters </div> <div> <div>Algorithm</div> <div>Breadth-First Search (BFS) </div> </div> <div> <div>Target Element</div> <div> Airplane </div> </div> <div> <div>Maximum Recipes</div> <div>-1</div> </div> <div> <div>Higher values will use multithreading for faster processing</div> <div>  </div> </div> <div> <div>Please enter a valid number for maximum recipes</div> </div> </div>

Analisis Hasil Pengujian

Dari hasil pengujian yang dilakukan terhadap aplikasi, dapat diambil hipotesis bahwa:

- Aplikasi berhasil menjalankan seluruh fitur utama dengan baik
- Fitur berjalan stabil baik pada algoritma Breadth-First Search (BFS) maupun Depth-First Search (DFS).

- Sistem dapat menangani input yang tidak valid, misalnya permintaan resep yang tidak mungkin, tanpa crash atau error fatal.
- UI mampu menampilkan status eksekusi seperti waktu, node yang dikunjungi, dan algoritma yang digunakan

Selanjutnya, dilakukan juga pengujian terhadap dua algoritma pencarian, yaitu DFS (Depth-First Search) dan BFS (Breadth-First Search) untuk menemukan *recipe* menuju elemen target "Alien", pada skala parameter maxRecipe yang berbeda: 1, 20, 50, dan 100. Semua pengujian menggunakan multithreading. Kecuali pada percobaan dengan maxRecipe bernilai 1.

Max Recipe	Algoritma	Nodes Visited	Execution Time (s)
1	DFS	15	0.109
1	BFS	13	0.092
20	DFS	145	1.181
20	BFS	131	0.702
50	DFS	203	0.549
50	BFS	397	0.548
100	DFS	405	0.611
100	BFS	755	0.588

Analisis hasil uji:

- Efisiensi Node: DFS secara konsisten mengunjungi lebih sedikit node dibandingkan BFS, terutama saat nilai maxRecipe membesar. Hal ini terlihat jelas pada maxRecipe = 100, di mana DFS hanya mengunjungi 405 node sementara BFS mengunjungi 755 node, menunjukkan keunggulan DFS dalam eksplorasi yang lebih hemat.
- Kecepatan Eksekusi: BFS cenderung lebih cepat dibanding DFS pada maxRecipe = 1, 20, dan 100, namun pada maxRecipe = 50, waktu eksekusi kedua algoritma hampir identik. Hal ini menunjukkan bahwa multithreading berperan besar dalam menjaga performa BFS tetap efisien, meskipun jumlah node yang dieksplorasi lebih banyak.
- Skalabilitas: DFS menunjukkan peningkatan skalabilitas yang baik, terlihat dari penurunan waktu eksekusi saat maxRecipe meningkat dari 20 ke 100 (dari 1.181s menjadi 0.611s). Sebaliknya, BFS tetap stabil dan cepat dalam waktu, tetapi dengan harga eksplorasi node yang jauh lebih tinggi, menunjukkan bahwa DFS menjadi lebih optimal untuk pencarian berskala besar.

BAB V: Kesimpulan dan Saran

5.1 Kesimpulan

Secara umum, BFS unggul dalam kecepatan eksekusi pada skala kecil hingga menengah, namun DFS lebih efisien dalam eksplorasi node pada skala besar dengan aplikasi

multithreading. Performa BFS tetap stabil karena multithreading yang membantu menangani eksplorasi node yang lebih banyak, sementara DFS efektif saat pencarian membutuhkan kedalaman. Pada saat maxRecipe bernilai 100, DFS mengunjungi hampir separuh node yang dijelajahi BFS, dengan waktu yang hampir setara, menjadikannya lebih unggul secara efisiensi.

5.2 Saran

Dalam pengembangan selanjutnya, dapat diimplementasikan strategi Bidirectional.

Lampiran

Aplikasi: littlealchemy2.vercel.app

Repositori Github Front-end: https://github.com/fathurwithyou/Tubes2_FE_RecipePlayground

Repositori Github Back-end: https://github.com/fathurwithyou/Tubes2_BE_RecipePlayground

Video Bonus Pengerjaan: <https://youtu.be/jbFKsliW0Ak>

Tabel I. Checklist Pengerjaan

No	Poin	Ya	Tidak
1	Aplikasi dapat dijalankan.	✓	
2	Aplikasi dapat memperoleh data recipe melalui scraping.	✓	
3	Algoritma Depth First Search dan Breadth First Search dapat menemukan recipe elemen dengan benar.	✓	
4	Aplikasi dapat menampilkan visualisasi recipe 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 Bidirectional.		✓
9	Membuat bonus Live Update.		✓
10	Aplikasi di-containerize dengan Docker.	✓	
11	Aplikasi di-deploy dan dapat diakses melalui internet.	✓	

Daftar Pustaka

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

React Flow API Reference. <https://reactflow.dev/api-reference>