LAPORAN TUGAS BESAR 2

MATA KULIAH IF2211 STRATEGI ALGORITMA

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

Dosen Pengampu:
Dr. Nur Ulfa Maulidevi, S.T, M.Sc.

Disusun Oleh:

Muhammad Aditya Rahmadeni	13523028
Nadhif Radityo Nugroho	13523045
Adhimas Aryo Bimo	13523052

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

MEI 2025

DAFTAR ISI

BAB I DESKRIPSI TUGAS **1**
1.1. Penjelasan Umum 1
BAB II LANDASAN TEORI **5**
2.1. Dasar Teori 5
2.2. Penjelasan Singkat Program 6
BAB III ANALISIS PEMECAHAN MASALAH **10**
3.1. Langkah-Langkah Pemecahan Masalah 10
3.2. Proses Pemetaan Masalah 10
3.3. Arsitektur Website Backend 11
3.4. Arsitektur Website Frontend 14
3.5. Bonus — Live Update 17
3.6. Bonus — Docker 19
3.7. Bonus — Deployment 21
3.8. Ilustrasi Kasus 22
BAB IV IMPLEMENTASI DAN PENGUJIAN **23**
4.1. Eksperimen 1 23
4.2. Eksperimen 2 24
4.3. Eksperimen 3 25
BAB V ANALISIS, KESIMPULAN, DAN SARAN **26**
5.1. Analisis 26
5.2. Kesimpulan 26
5.3. Saran 27
BAB VI LAMPIRAN **28**
6.1. Tautan 28
6.2. Tabel Evaluasi Fungsionalitas 28

BAB I DESKRIPSI TUGAS

1.1. Penjelasan Umum

Gambar 1. Little Alchemy 2
(Sumber: https://www.thegamer.com)

Little Alchemy 2 merupakan permainan berbasis web / aplikasi yang dikembangkan oleh Recloak yang dirilis pada tahun 2017, permainan ini bertujuan untuk membuat 720 elemen dari 4 elemen dasar yang tersedia yaitu air, earth, fire, dan water. Permainan ini merupakan sekuel dari permainan sebelumnya yakni Little Alchemy 1 yang dirilis tahun 2010.

Mekanisme dari permainan ini adalah pemain dapat menggabungkan kedua elemen dengan melakukan drag and drop, jika kombinasi kedua elemen valid, akan memunculkan elemen baru, jika kombinasi tidak valid maka tidak akan terjadi apa-apa. Permainan ini tersedia di web browser, Android atau iOS

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

Komponen-komponen dari permainan ini antara lain:

1. Elemen dasar

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

Gambar 2. Elemen dasar pada Little Alchemy 2

2. Elemen turunan

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

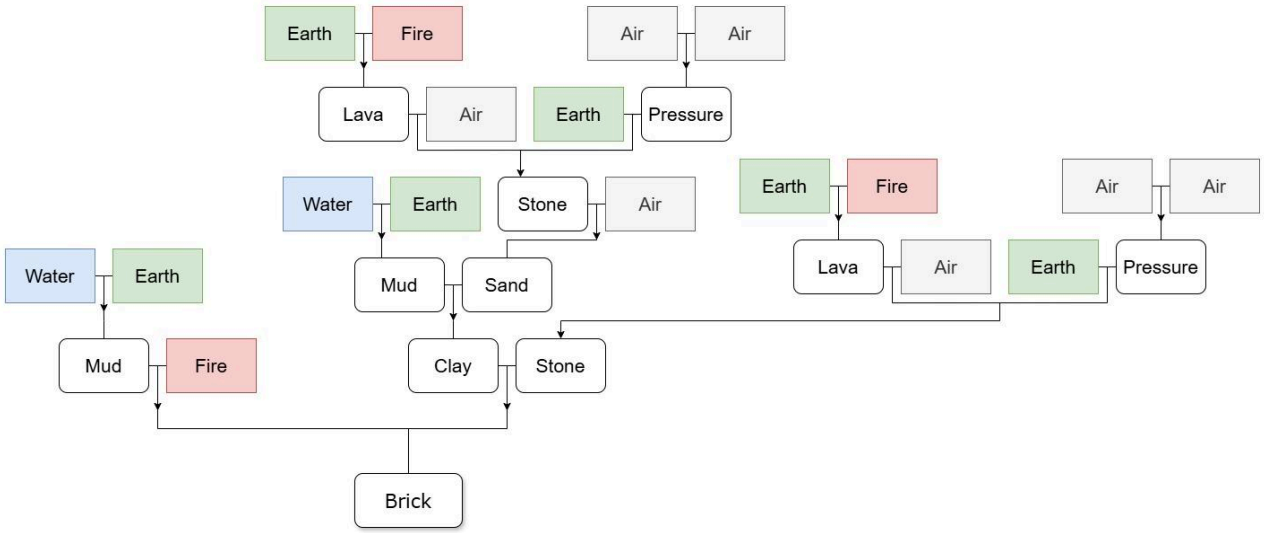
3. Combine Mechanism

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

Spesifikasi Wajib:

- Buatlah aplikasi pencarian recipe elemen dalam permainan Little Alchemy 2 dengan menggunakan strategi BFS dan DFS.
- Tugas dikerjakan berkelompok dengan anggota minimal 2 orang dan maksimal 3 orang, boleh lintas kelas dan lintas kampus.
- Aplikasi berbasis web, untuk frontend dibangun menggunakan bahasa Javascript dengan framework Next.js atau React.js, dan untuk backend menggunakan bahasa Golang.

- Untuk repository frontend dan backend diperbolehkan digabung maupun dipisah.
- Untuk data elemen beserta resep dapat diperoleh dari scraping website Fandom Little Alchemy 2.
- Terdapat opsi pada aplikasi untuk memilih algoritma BFS atau DFS (juga bidirectional jika membuat bonus)
- Terdapat toggle button untuk memilih untuk menemukan sebuah recipe (Silahkan yang mana saja) terpendek (output dengan rute terpendek) atau mencari banyak recipe (multiple recipe) menuju suatu elemen tertentu. Apabila pengguna ingin mencari banyak recipe maka terdapat cara bagi pengguna untuk memasukkan parameter banyak recipe maksimal yang ingin dicari. Aplikasi boleh mengeluarkan recipe apapun asalkan berbeda dan memenuhi banyak yang diinginkan pengguna (apabila mungkin).
- Mode pencarian multiple recipe wajib dioptimasi menggunakan multithreading.
- Elemen yang digunakan pada suatu recipe harus berupa elemen dengan tier lebih rendah dari elemen yang ingin dibentuk.
- Aplikasi akan memvisualisasikan recipe yang ditemukan sebagai sebuah tree yang menunjukkan kombinasi elemen yang diperlukan dari elemen dasar. Agar lebih jelas perhatikan contoh berikut:



Gambar 3. Contoh visualisasi recipe elemen

Gambar diatas menunjukkan contoh visualisasi recipe dari elemen Brick. Setiap elemen bersebelahan menunjukkan elemen yang perlu dikombinasikan. Amati bahwa leaf

dari tree selalu berupa elemen dasar. Apabila dihitung, gambar diatas menunjukkan 5 buah recipe untuk Brick (karena Brick dapat dibentuk dengan kombinasi Mud+Fire atau Clay+Stone, begitu pula Stone yang dapat dibentuk oleh kombinasi Lava+Air atau Earth+Pressure). Visualisasi pada aplikasi tidak perlu persis seperti contoh diatas, tetapi pastikan bahwa recipe ditampilkan dengan jelas. Aplikasi juga menampilkan waktu pencarian serta banyak node yang dikunjungi.

BAB II LANDASAN TEORI

2.1. Dasar Teori

Graf adalah struktur data yang terdiri dari simpul (node/vertex) dan sisi (edge) yang menghubungkan antar simpul. Graf dapat berupa berarah (directed) atau tak berarah (undirected), dan bisa memiliki berat (weighted) atau tidak.

Penjelajahan graf (Graph Traversal) adalah proses mengunjungi simpul-simpul dalam graf untuk mencari informasi atau menyelesaikan suatu permasalahan (misalnya mencari jalur, komponen, atau mengecek keterhubungan).

Dua metode penjelajahan graf yang paling umum adalah:

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

   Breadth First Search (BFS) adalah algoritma penelusuran graf yang bekerja dengan prinsip penelusuran level demi level dari simpul awal. BFS menggunakan pendekatan First In First Out (FIFO) dengan struktur data queue (antrian) untuk memastikan bahwa simpul yang ditemukan lebih awal diproses terlebih dahulu.

   Mekanisme Algoritma BFS ;

   1. BFS dimulai dari simpul awal (source node), yang langsung ditandai sebagai dikunjungi dan dimasukkan ke dalam queue.
   2. *Loop* dimulai dengan kondisi *queue* tidak kosong
   3. Ambil simpul pertama dari *queue*.
   4. Periksa seluruh simpul tetangga yang ada. Jika simpul belum diproses, masukkan simpul tetangga ke dalam *queue*.

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

   Depth First Search (DFS) adalah algoritma penelusuran graf yang bekerja dengan prinsip penelusuran mendalam terlebih dahulu. Alih-alih menjelajahi simpul pada level yang sama, DFS terus bergerak ke satu cabang sedalam mungkin sebelum backtrack.

   DFS dapat diimplementasikan dengan:

   - Struktur data stack (Last In First Out – LIFO).
   - Pendekatan rekursif, yang secara implisit menggunakan stack fungsi pemanggilan.

Mekanisme Algoritma DFS;

1. Mulai dari simpul awal, tandai sebagai dikunjungi.
2. Lanjutkan ke tetangga pertama yang belum dikunjungi.
3. Ulangi proses hingga mencapai simpul akhir atau tidak ada tetangga yang belum dikunjungi.
4. Jika buntu (dead end), kembali (backtrack) ke simpul sebelumnya, dan lanjutkan pencarian.

2.2. Penjelasan Singkat Program

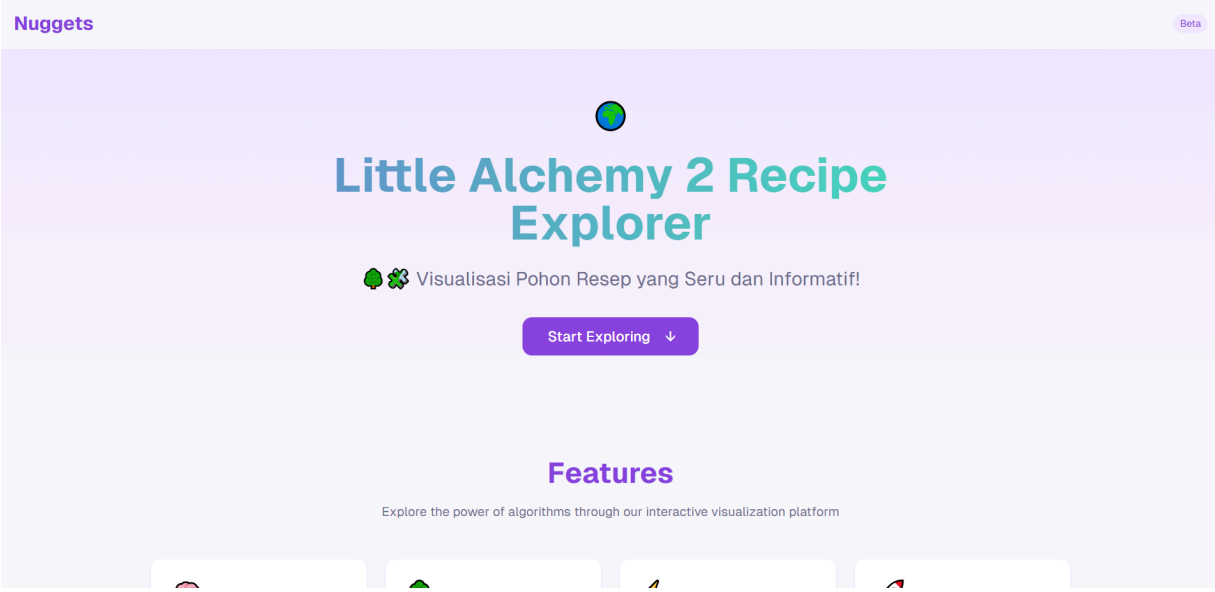
Pada tugas besar kali ini, kelompok kami membuat program untuk mencari resep dari sebuah elemen menggunakan metode pencarian graf yang dijelaskan pada 2.1. Hal tersebut dapat dilakukan dengan meletakkan hipotesis awal bahwa semua elemen dapat terbentuk dari elemen dasar dan setiap elemen memiliki resep untuk membuat elemen tersebut.

Pada kasus ini, kami menerapkan algoritma BFS dan DFS menggunakan bahasa pemrograman Go dan menampilkan hasil resep kepada pengguna pada aplikasi berbasis website menggunakan bahasa pemrograman TypeScript. Berikut struktur directory kelompok kami:

```
Tubes2_Nuggets/
├── .vscode/
├── doc/
├── src/
│   ├── backend/
│   │   ├── element-api/
│   │   │   └── server.go
│   │   ├── scrap/
│   │   │   ├── elements.json
│   │   │   └── scrap.go
│   │   └── utils/
│   ├── .dockerignore
│   ├── .gitignore
│   ├── Dockerfile
│   ├── go.mod
│   ├── go.sum
│   └── main.go
├── frontend/
│   ├── app/
│   │   ├── globals.scss
│   │   ├── layout.tsx
```

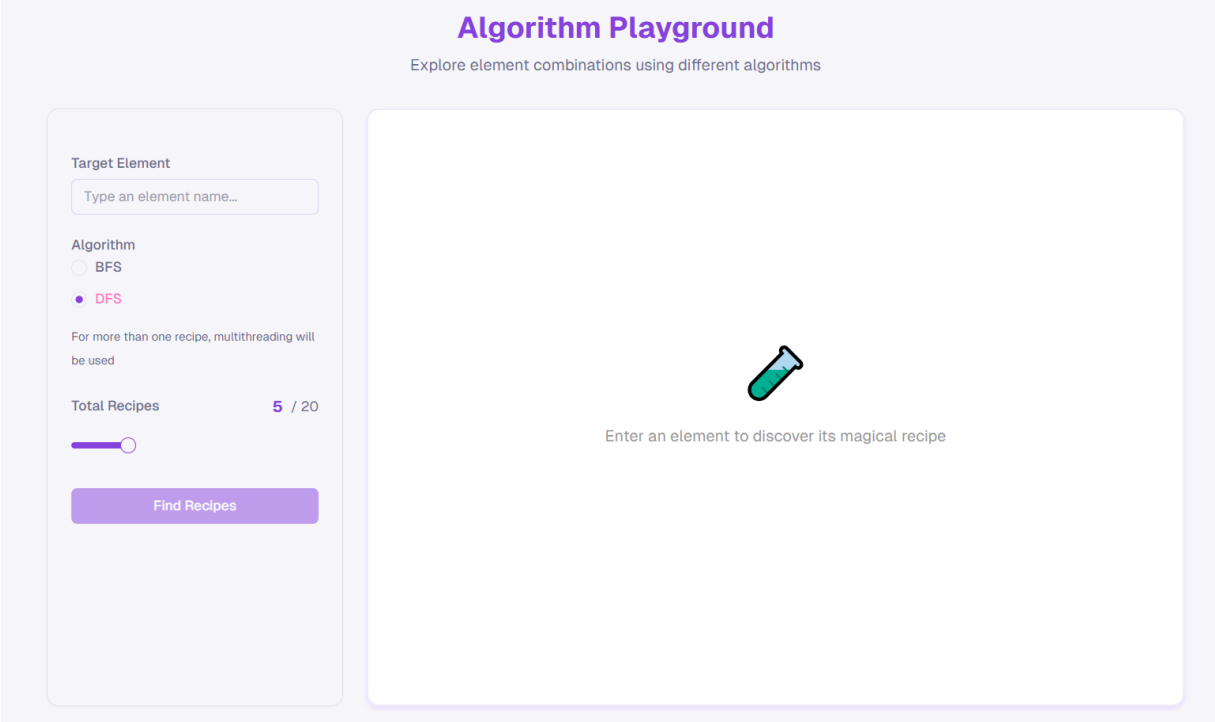
```
│   │   └── page.tsx
│   ├── components/
│   │   ├── algorithm-controls.tsx
│   │   ├── d3-visualization.tsx
│   │   ├── features.tsx
│   │   ├── hero.tsx
│   │   ├── motion-wrapper.tsx
│   │   └── theme.ts
│   ├── lib/
│   │   └── api/
│   │       └── utils.ts
│   ├── public/
│   │   ├── logo-circle.png
│   │   ├── logo.png
│   │   └── test.html
│   ├── .dockerignore
│   ├── .gitignore
│   ├── Dockerfile
│   ├── components.json
│   ├── eslint.config.mjs
│   ├── next-env.d.ts
│   ├── next.config.mjs
│   ├── package-lock.json
│   ├── package.json
│   ├── postcss.config.mjs
│   ├── tailwind.config.js
│   ├── tsconfig.json
│   ├── .gitkeep
│   └── docker-compose.yml
├── .gitignore
└── README.md
```

Hasil program kami telah dirilis dalam website dengan pranala berikut: https://nuggets.bwks.link/. Untuk memudahkan, dapat dilihat tampilan website secara sekilas pada gambar dibawah ini.



Gambar 4. Tampilan Awal Website

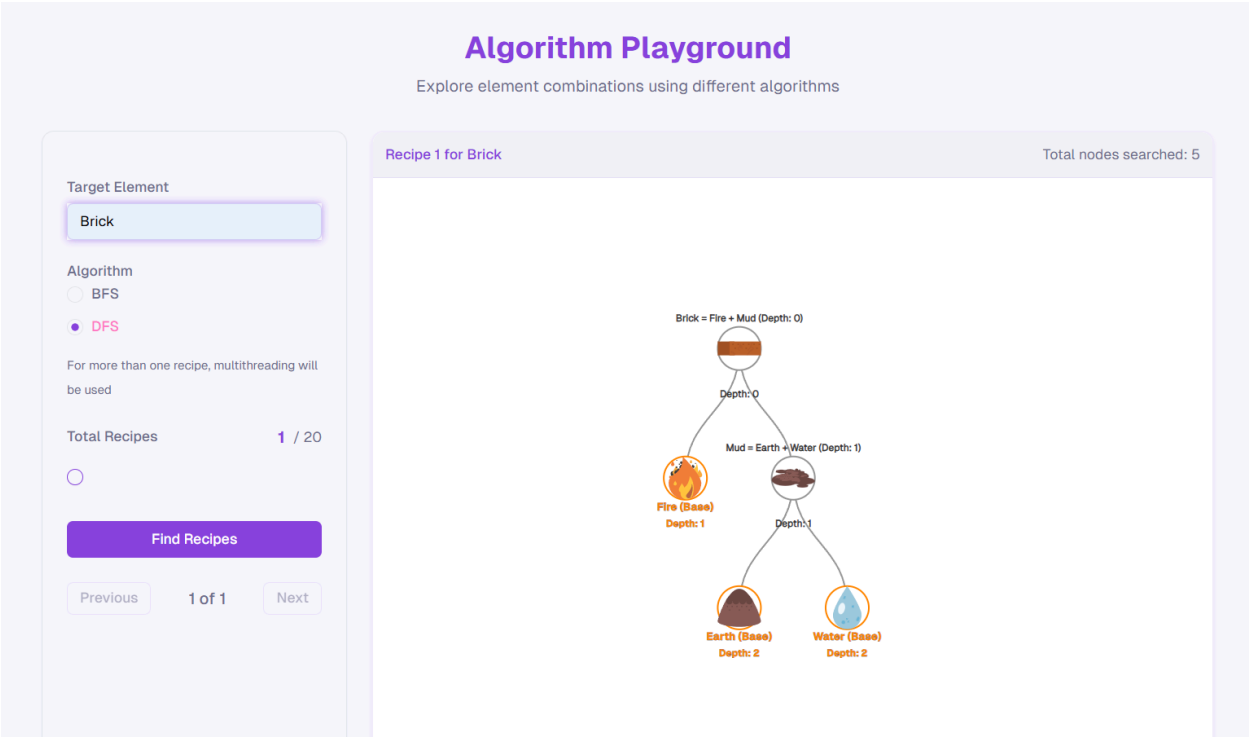
Tampilan awal ini bertujuan untuk memberikan keterangan kepada pengguna intensi pembuatan website ini, yakni mencari resep pada sebuah permainan Little Alchemy 2. Selanjutnya pengguna dapat menekan tombol "Start Exploring" untuk melakukan pencarian resep menggunakan algoritma BFS atau DFS.



Gambar 5. Tampilan Pencarian Resep

Pada tampilan ini, pengguna dapat melakukan pencarian elemen yang ada pada Little Alchemy 2. Pencarian juga dapat dilakukan dengan memilih algoritma sesuai dengan keinginan pengguna. Untuk jumlah pencarian resep, kami batasi hingga 20 agar tidak terlalu lama dalam melakukan pencarian dan selama melakukan eksperimen belum ada elemen yang dapat memiliki lebih dari 20 alur resep berbeda.

Ketika melakukan pencarian multi resep, akan dilakukan multithreading untuk mempercepat proses pencarian. Multithreading ini diterapkan baik untuk DFS maupun BFS. Selanjutnya hasil resep dapat dilihat pada kanvas sebelah kanan. Berikut merupakan contoh hasil pencarian.



Gambar 6. Hasil Tampilan Resep

Resep elemen yang tertampil pada kanvas dan alur resep elemen akan ditemukan hingga mencapai ke bagian elemen dasar. Selain itu, terlihat beberapa atribut untuk benchmarking seperti jumlah node yang dikunjungi pada bagian atas dan waktu yang diperlukan untuk mencari resep tersebut. Untuk melihat resep lain, dapat menekan tombol “Next”.

BAB III ANALISIS PEMECAHAN MASALAH

3.1. Langkah-Langkah Pemecahan Masalah

Sebelum menerapkan persoalan ini, tentunya perlu untuk memahami kunci utama dalam permainan. Kunci utama tersebut adalah elemen dapat dibentuk menggunakan 2 elemen lainnya. Dengan penjelasan tersebut ada 2 cara yang terpikirkan bagi kelompok kami dalam melakukan pencarian resep.

1. Melakukan pencarian dari elemen dasar
2. Melakukan pencarian dari elemen target

Kedua pencarian tersebut dapat dilakukan. Jika melakukan pencarian dari elemen dasar maka akan melakukan kombinasi dari berbagai elemen tersebut hingga mencapai target, sedangkan untuk pencarian selanjutnya hanya mencari elemen yang menjadi bahan untuk resep target dan melakukan pencarian hingga sampai ke elemen dasar.

Oleh karena itu, kelompok kami memilih pendekatan kedua karena tidak perlu melakukan pencarian dari awal untuk setiap kemungkinan. Pendekatan ini lebih efisien, karena cukup menelusuri jalur yang relevan dari elemen target menuju elemen dasar.

3.2. Proses Pemetaan Masalah

Untuk mengimplementasikan pendekatan pencarian resep dari elemen target, pertama-tama kita perlu memahami bagaimana cara menyusun dan menelusuri pohon pencarian. Pohon ini akan memetakan hubungan antara elemen-elemen yang dapat saling membentuk melalui resep.

1. Menjadikan Elemen Target sebagai Root pada Pohon Pencarian

   Setiap elemen target memiliki resep yang terdiri dari dua elemen yang dapat digunakan untuk membentuknya. Elemen target akan menjadi root dalam pohon pencarian ini, dan kita akan mencari bahan-bahan yang membentuk elemen tersebut.

   Setiap simpul dalam pohon ini akan mewakili elemen yang terlibat dalam pembentukan elemen target, dengan cabang yang menunjukkan elemen yang membentuknya.

2. Penggunaan Tier untuk Mencegah Looping

Tier adalah konsep yang membatasi kedalaman pencarian dalam pohon resep. Setiap elemen memiliki Tier yang menunjukkan jaraknya dari elemen dasar. Elemen dengan Tier yang lebih rendah berarti lebih dekat dengan elemen dasar, sedangkan elemen dengan Tier lebih tinggi berarti lebih kompleks dan lebih jauh dari elemen dasar.

Dalam pemilihan resep, kita hanya akan memilih elemen dengan Tier yang lebih rendah sebagai bahan pembentuk resep. Ini penting untuk menghindari looping atau pencarian berulang yang bisa terjadi jika elemen dengan Tier lebih tinggi dipilih, yang dapat menyebabkan pencarian menjadi tidak efisien.

3. Penerapan Algoritma pada Pencarian Resep

   Implementasi algoritma pencarian akan dilakukan dengan menelusuri pohon resep berdasarkan urutan tier dan memilih resep yang sesuai dengan kondisi yang ada (apakah menggunakan BFS atau DFS).

   Pada implementasi BFS, pencarian dimulai dengan level pertama, mencari resep untuk elemen yang lebih dekat ke elemen dasar. Setelah level pertama selesai, baru dilanjutkan ke level berikutnya.

   Pada implementasi DFS, kita akan menelusuri resep hingga mencapai solusi, dan jika diperlukan, kembali ke level sebelumnya untuk mencoba resep lainnya.

3.3. Arsitektur Website Backend

Pada backend, website menggunakan bahasa pemrograman Go, sesuai yang diperintahkan oleh spesifikasi. Dengan beberapa algoritma terdapat dalam direktori khusus untuk mempermudah manajemen kode. Dan untuk kode entry server terdapat di suatu file khusus. Dengan ini, semua bagian kode dapat berinteraksi secara langsung dan efisien.

Untuk live update, website kami juga memanfaatkan WebSocket sebagai media komunikasi realtime. Dengan ini, semua progres step dari algoritma yang dipilih dapat ditampilkan di frontend secara langsung.

server.go 47ff124fc49a897a65aeafe3d33551f761cba67c L194-L260

```go
func findPath(c *gin.Context) {
	// Get query parameters
	algorithm := c.Query("algorithm")
```

```
	direction := c.Query("direction")
	count, err := strconv.Atoi(c.Query("count"))
	if err ≠ nil {
		c.JSON(http.StatusBadRequest, gin.H{ "error": "Invalid
count" })
		return
	}
	targetElement := c.Query("target")

	// Validate parameters
	if algorithm ≠ "dfs" && algorithm ≠ "bfs" && algorithm ≠
"bfs-shortest" {
		c.JSON(http.StatusBadRequest, gin.H{ "error": "Invalid
algorithm" })
		return
	}
	if direction ≠ "source" && direction ≠ "target" {
		c.JSON(http.StatusBadRequest, gin.H{ "error": "Invalid
direction" })
		return
	}
	if count == 0 {
		c.JSON(http.StatusBadRequest, gin.H{ "error": "Count is
required" })
		return
	}

	target, targetExists := cachedRecipesEl[targetElement]
	if !targetExists {
	  c.JSON(http.StatusBadRequest, gin.H{ "error": "Target element
not found" })
	  return
    }

	// Upgrade to WebSocket
	wsConn, err := upgrader.Upgrade(c.Writer, c.Request, nil)
	if err ≠ nil {
		c.AbortWithStatusJSON(http.StatusInternalServerError,
gin.H{ "error": "WebSocket upgrade failed" })
		return
	}
	defer wsConn.Close()

	resultChan := make(chan utils.Message)

	go func() {
		if algorithm == "dfs" && direction == "target" {
			// Perform DFS from target to source
```

```
                    if count == 1 {
                          utils.DFS(cachedRecipesMap, cachedRecipesEl,
target, resultChan)
                    } else {
                          utils.DFSMultiple(cachedRecipesMap,
cachedRecipesEl, target, int(count), resultChan)
                    }
              }
              if algorithm == "bfs" && direction == "target" {
                    // Perform BFS from target to source
                    if count == 1 {
                          utils.BFSShortestPath(targetElement,
cachedRecipesMap, cachedRecipesEl, resultChan)
                    } else {
                          utils.BFSP(targetElement, cachedRecipesMap,
cachedRecipesEl, int(count), resultChan)
                    }
              }
              close(resultChan)
        }()

        for wsMsg := range resultChan {
              if err := wsConn.WriteJSON(wsMsg); err ≠ nil {
                    break;
              }
        }
}
```

Perlu juga untuk algoritma-algoritma BFS dan DFS untuk dapat mengeluarkan hasil parsialnya sehingga dapat live update tanpa menunggu hasil lainnya untuk selesai juga. Sebagai contoh:

dfs_multi.go 47ff124fc49a897a65aeafe3d33551f761cba67c L11-L41

```
func DFSMultiple(recipeMap RecipeMap, recipesEl RecipeElement,
targetElement Element, maxPaths int, resultChan chan Message) {
    var mutex sync.Mutex
    var wg sync.WaitGroup
    resultSet := make(map[string]bool)
    numWorkers := min(maxPaths * 2, 10)
    wg.Add(numWorkers)
    start := time.Now().UnixNano()

    for i := 0; i < numWorkers; i++ {
        go func(workerID int) {
            defer wg.Done()
            seed := workerID
```

```
            visited := make(map[string]bool)
            nodesVisited := 0
            recipePath := DFSHelperWithVariation(recipeMap, recipesEl,
targetElement.Name, visited, &nodesVisited, 0, seed)
            pathSignature := generatePathSignature(recipePath)

            mutex.Lock()
            if len(recipePath) > 0 && len(resultSet) < maxPaths &&
!resultSet[pathSignature] {
                resultSet[pathSignature] = true
                resultChan ← Message{
                    RecipePath: recipePath,
                    NodesVisited: nodesVisited,
                    Duration: float32(time.Now().UnixNano() - start) /
1000000,
                }
            }
            mutex.Unlock()
        }(i)
    }
    wg.Wait()
}
```

3.4. Arsitektur Website Frontend

Dengan arsitektur backend yang telah didesain, maka diperlukan juga sarana utilitas agar frontend dapat berkomunikasi dengan backend secara efisien. Dalam satu koneksi WebSocket, client hanya akan mengirimkan parameternya melalui query parameters. Namun, setelah itu client tidak akan lagi mengirim data melalui WebSocket, sebaliknya server akan mengirimkan data hasil algoritma pencarian dan client akan memprosesnya untuk ditampilkan ke user.

lib/api/service.tsx 47ff124fc49a897a65aeafe3d33551f761cba67c L36-L66

```
export function createRecipeWebSocket(
  algorithm: 'dfs' | 'bfs' | 'bfs-shortest',
  direction: 'target',
  targetElement: string,
  count: number,
  onMessage: (result: AlgorithmResponse) ⇒ void,
  onError: (error: Event) ⇒ void
): WebSocket {
  const url = new URL("/api/find-recipe",
process.env.NEXT_PUBLIC_BACKEND);
```

```
  url.searchParams.set("algorithm", algorithm);
  url.searchParams.set("direction", direction);
  url.searchParams.set("target", targetElement);
  url.searchParams.set("count", `${count}`);
  const socket = new WebSocket(url);
  const startTime = performance.now();

  socket.onmessage = (event) => {
    try {
      const data = JSON.parse(event.data);
      if(typeof data.duration ≠ "number")
        data.duration = performance.now() - startTime;
      onMessage(data);
    } catch (err) {
      console.error('Error parsing WebSocket message:', err);
    }
  };

  socket.onerror = onError;

  return socket;
}
```

Untuk proses penggambaran visualisasi, website menggunakan framework D3.js. D3.js menawarkan utilitas penggambaran graf yang versatile atau serba guna, sehingga sangat cocok dengan kebutuhan dalam website ini. Setiap kali terdapat data baru yang masuk, D3.js akan membuat tree baru. Namun pada saat update elemen terhadap tree yang baru, D3.js akan melakukan diferensiasi data. Sehingga, Document Object Model (DOM) yang diedit tidak terlalu banyak dan sangat efisien.

d3-visualization.tsx 47ff124fc49a897a65aeafe3d33551f761cba67c L155-L273

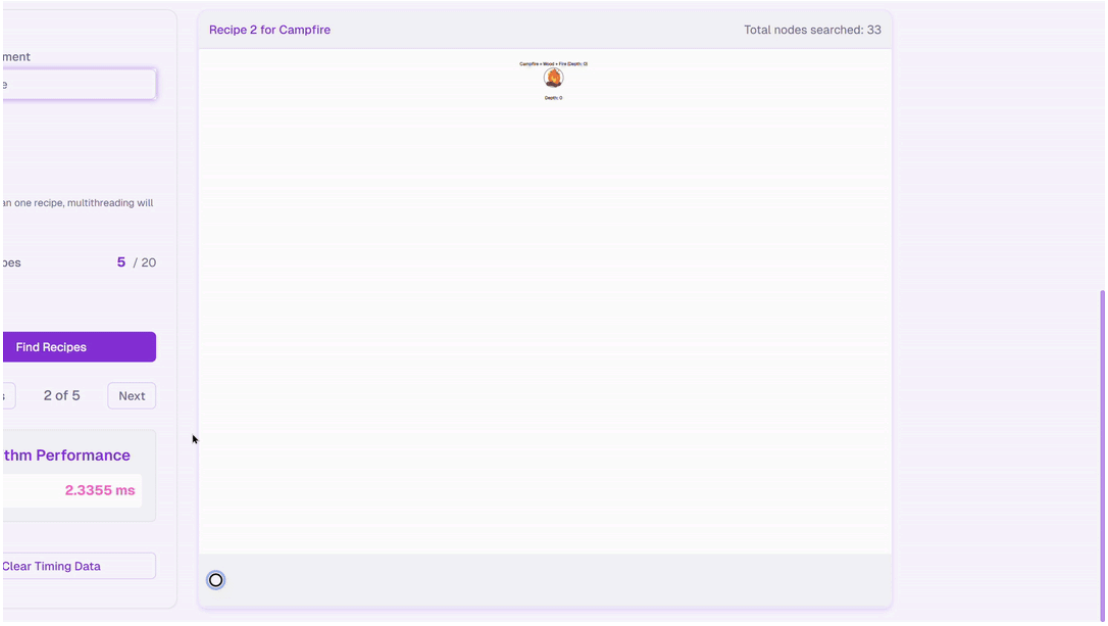
```
const treeData = buildTreeNode(targetElement);

// Create d3 hierarchy
const root = d3.hierarchy(treeData);

// Create tree layout
const treeLayout = d3.tree<TreeNode>()
  .nodeSize([100, 120]); // [horizontal, vertical] spacing

// Apply the layout
const hierarchyData = treeLayout(root);

// Create container group and center it
```

```
const container = d3.select(containerRef.current);

// Center the tree horizontally
const descendants = hierarchyData.descendants();
const minX = d3.min(descendants, d => d.x) || 0;
const maxX = d3.max(descendants, d => d.x) || 0;
const centerX = (maxX + minX) / 2;

// Transform to center the tree
container.attr("transform", `translate(${width / 2 - centerX}, 50)`);

// Create container for links
const linkGroup = container.append("g").attr("class", "links");

// Draw links
linkGroup.selectAll("path")
  .data(hierarchyData.links())
  .enter()
  .append("path")/* ... snip ... */
  .attr("d", d3.linkVertical<any, any>()
    .x(d => d.x)
    .y(d => d.y)
  );

// Create container for nodes
const nodeGroup = container.append("g").attr("class", "nodes");

// Create node groups
const nodes = nodeGroup.selectAll(".node")
  .data(hierarchyData.descendants())
  .enter()
  .append("g")/* ... snip ... */;

// Add white background circle for better visibility
nodes.append("circle")/* ... snip ... */;

// Add element images
nodes.append("image")
  .attr("xlink:href", d => new URL(`/images/${d.data.name}.svg`,
process.env.NEXT_PUBLIC_BACKEND).href)/* ... snip ... */
  .on("error", function() {
    // Fallback for failed images
    const d = d3.select(this).datum() as any;
    d3.select(this)
      .attr("xlink:href",
`https://placehold.co/40×40/orange/white?text=${d.data.name.charAt(0)}
`);
  });
```

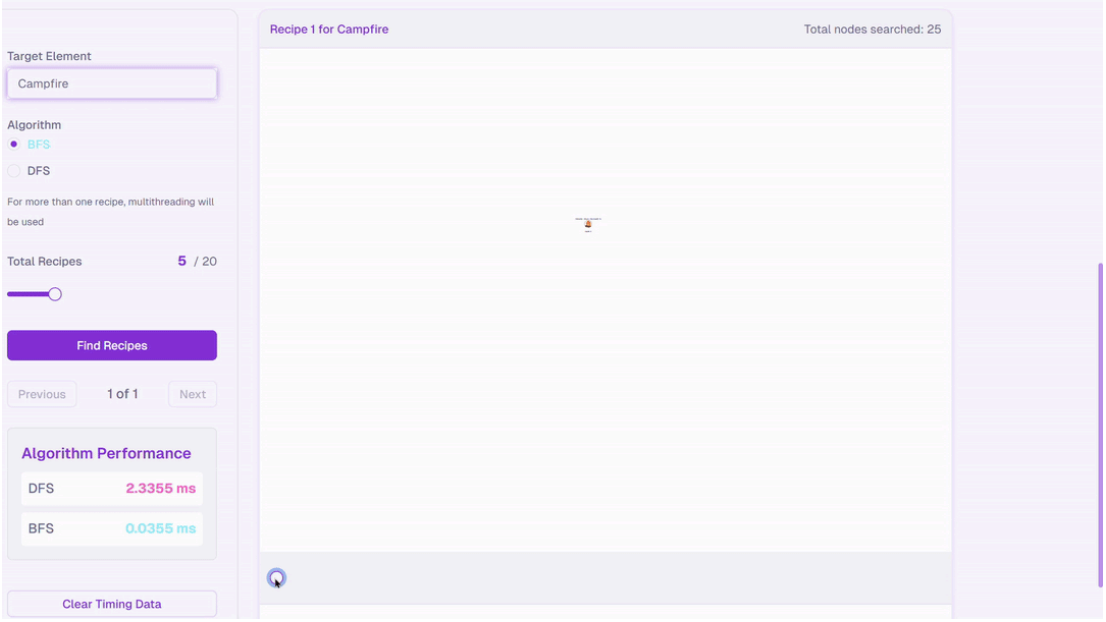
```
// Add element name labels
nodes.filter(d => d.data.path.ingredient1 == "")
  .append("text")
  .attr("dy", 30)/* ... snip ... */
  .text(d => {
    const recipePath = d.data.path;
    // Show "Base" suffix for base elements
    return recipePath.ingredient1 == "" && recipePath.ingredient2 ==
"" ?
      `${recipePath.result} (Base)` : recipePath.result;
  });

// Add recipe formula and depth (for non-base elements)
nodes.filter(d => d.data.path.ingredient1 != "")
  .append("text")/* ... snip ... */;
container.attr("transform", zoomTransformRef.current);
```

3.5. Bonus — Live Update

Secara teknis, hasil dari tiap-tiap algoritma BFS atau DFS telah di-stream secara real-time ke frontend. Namun, karena data yang masuk begitu cepat, dan setelah pertimbangan performansi rendering frontend juga held-connection backend, diimplementasikan sebuah slider yang dapat menunjukkan progres algoritma DFS ataupun BFS. Secara konsep, pada DFS akan menelusuri node terdalam terlebih dahulu. Sebaliknya, untuk BFS akan menelusuri node secara luas terlebih dahulu. Ini dapat dibuktikan pada tangkapan layar di bawah:



Gambar 7. Progresif Live Update Algoritma DFS
(gambar adalah animasi, buka dokumen untuk melihat animasi)



Gambar 8. Progresif Live Update Algoritma BFS
(gambar adalah animasi, buka dokumen untuk melihat animasi)

3.6. Bonus — Docker

Dockerfile menggunakan pendekatan multi-stage build untuk membuat image aplikasi Go yang ringan dan efisien. Pada tahap pertama (builder), image dasar golang:1.24-alpine digunakan untuk membangun aplikasi dengan menonaktifkan CGO dan menargetkan arsitektur Linux amd64. Dependensi diunduh melalui go mod download, lalu seluruh kode disalin dan dikompilasi menjadi binary server. Pada tahap kedua (runner), hasil build dari tahap pertama disalin ke image alpine:latest yang minimalis, *working directory* diatur, port 8888 diekspos, dan perintah default dijalankan untuk mengeksekusi binary server.

backend/Dockerfile ad6ba6f5743eba0f5fb7819699f26302b3a58b84

```
FROM golang:1.24-alpine AS builder
ENV CGO_ENABLED=0 GOOS=linux GOARCH=amd64
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN go build -o element-api/server element-api/server.go

FROM alpine:latest AS runner
WORKDIR /app
COPY --from=builder /app/element-api/server ./element-api/server
VOLUME /app/scrap
EXPOSE 8888
WORKDIR /app/element-api
CMD ["./server"]
```

Melanjutkan pendekatan efisien dari Dockerfile sebelumnya untuk aplikasi Go, Dockerfile ini menerapkan prinsip serupa dalam membangun aplikasi frontend berbasis Next.js menggunakan multi-stage build untuk menjaga ukuran image tetap kecil dan terstruktur. Tahap pertama (dependencies) digunakan untuk menginstal dependensi menggunakan npm ci. Pada tahap kedua (builder), kode proyek disalin beserta modul Node.js yang telah terinstal, lalu proses build dilakukan dengan mengatur variabel lingkungan NEXT_PUBLIC_BACKEND yang mengarah ke backend Go

(http://localhost:8888), sehingga menunjukkan keterhubungan antara frontend dan backend. Tahap terakhir (runner) hanya menyalin hasil build dan file yang dibutuhkan ke dalam image minimalis berbasis node:24-alpine, mengekspos port 3000, dan menjalankan aplikasi dalam mode produksi.

frontend/Dockerfile ad6ba6f5743eba0f5fb7819699f26302b3a58b84

```
FROM node:24-alpine AS dependencies
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci

FROM node:24-alpine AS builder
WORKDIR /app
COPY --from=dependencies /app/node_modules ./node_modules
COPY . .
ENV NEXT_PUBLIC_BACKEND=http://localhost:8888
RUN npm run build

FROM node:24-alpine AS runner
WORKDIR /app
COPY --from=builder /app/public ./public
COPY --from=builder /app/.next ./.next
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/package.json ./package.json
ENV NODE_ENV=production
EXPOSE 3000
CMD ["npm", "start"]
```

File docker-compose.yml ini mengoordinasikan layanan frontend dan backend dalam satu sistem terintegrasi. Frontend dibangun dari folder ./frontend, terhubung ke backend melalui env variable, dan menunggu backend siap sebelum dijalankan. Backend dibangun dari ./backend, mengekspos port 8888, dan menggunakan volume untuk penyimpanan data lokal. Keduanya dikonfigurasi agar otomatis restart jika berhenti.

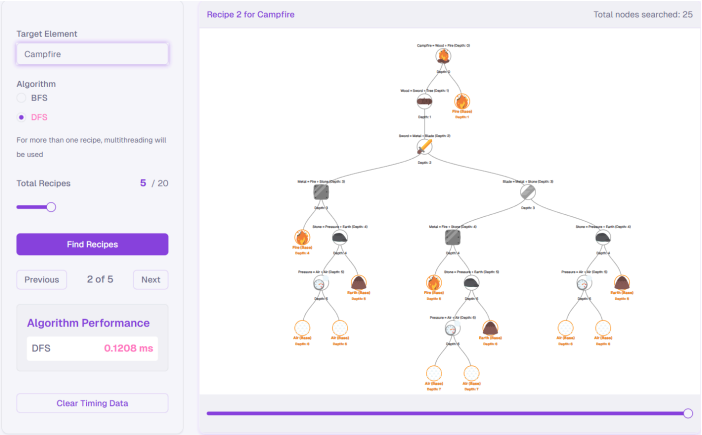
docker-compose.yml ad6ba6f5743eba0f5fb7819699f26302b3a58b84

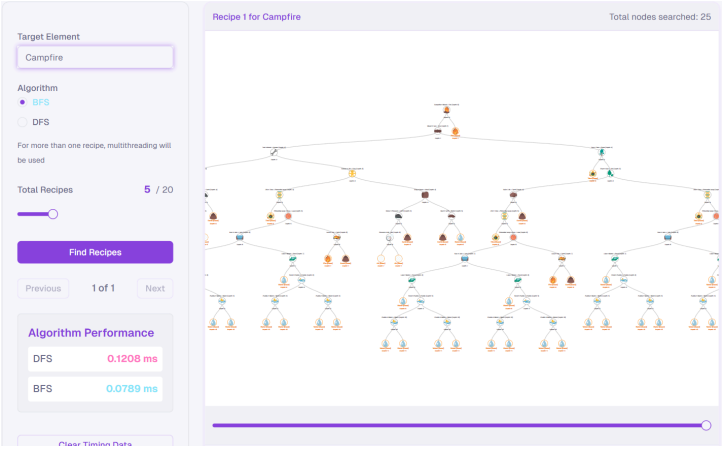
```yaml
version: '3.8'

services:
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    depends_on:
      - backend
    environment:
      - NEXT_PUBLIC_BACKEND=http://localhost:8888
    restart: unless-stopped
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    volumes:
      - ./src/backend/scrap/:/app/scrap/
    ports:
      - "8888:8888"
    restart: unless-stopped
```

3.7. Bonus — Deployment

Website di-deploy secara publik di suatu VM Server on-prem pribadi dengan alamat domain https://nuggets.bwks.link. Server menjalankan OS Windows Server 2022 Datacenter dengan 4 cores CPU, 4GB RAM, 92 GB Disk. Routing menggunakan cloudflare tunnel beserta DNS servicenya. Domain milik pribadi. Rencana uptime selama kurang lebih 3 bulan.

3.8. Ilustrasi Kasus





Struktur resep *Campfire* dalam Little Alchemy 2 dapat dianalisis menggunakan algoritma pencarian seperti *Depth-First Search* (DFS) dan *Breadth-First Search* (BFS), di mana DFS mencerminkan proses menelusuri jalur kombinasi sedalam mungkin—misalnya dari *Campfire* → *Wood* → *Sword* → *Blade* → *Stone* → *Lava* → *Earth*—sebelum kembali dan mencoba jalur lain, sedangkan BFS akan mengeksplorasi semua kombinasi pada level terdekat terlebih dahulu, seperti mencoba semua kemungkinan hasil dari elemen dasar (*Fire*, *Earth*, *Air*) sebelum naik ke kedalaman berikutnya. Dalam contoh ini, DFS cocok untuk menjelajahi seluruh struktur resep secara menyeluruh, sementara BFS lebih efisien untuk menemukan jalur kombinasi terpendek dari bahan dasar ke *Campfire*.

BAB IV IMPLEMENTASI DAN PENGUJIAN

4.1. Struktur Data

Untuk mempermudah pengambilan elemen dari hasil *scraping*, Beberapa struktur data dibentuk sebagai struct kru yang akan diproses pada algoritma BFS dan DFS nanti dan sebagai struktur yang akan diterima oleh *front end* nantinya.

elements.go

```
type RecipePath struct {
    Ingredient1 string `json:"ingredient2"`
    Ingredient2 string `json:"ingredient1"`
    Result      string `json:"result"`
}
type Element struct {
    Name     string      `json:"name"`
    Recipes  [][2]string `json:"recipes"`
    Image    string      `json:"image"`
    PageURL  string      `json:"page_url"`
    Tier    int          `json:"tier"`
}
type Message struct {
    RecipePath []RecipePath `json:"recipePath"`
    NodesVisited int        `json:"nodesVisited"`
    Duration float32        `json:"duration"`
}
```

4.2. Implementasi Algoritma BFS

Implementasi algoritma BFS sesuai dengan teori, yaitu menggunakan queue sebagai urutan pemrosesan lalu memasukkan setiap elemen yang valid ke dalam antrian. Jika mencapai elemen dasar maka pohon akan di-cek apakah penuh atau tidak

bfs.go

```
package utils

import (
```

```
    "sort"
    "time"
)

type Node struct {
    Step int
    Ingredient1 *Node
    Ingredient2 *Node
    Result      string
}

func BFSShortestNodeImpl(target string, Elements
map[string]Element, visitedNode *int) (*Node) {
    step := 0
    root := &Node{ Result: target, Step: step }
    nodeMap := map[string]*Node{ target: root }
    queue := []*Node{ root }

    for len(queue) > 0 {
        current := queue[0]
        queue = queue[1:]
        *visitedNode++

        // Jika base element, tidak perlu proses resep
        if BaseElement[current.Result] {
            continue
        }

        elem, ok := Elements[current.Result]
        if !ok {
            continue
        }

        for _, recipe := range elem.Recipes {
            ing1 := recipe[0]
            ing2 := recipe[1]

            // Cek ingredient valid dan tier lebih kecil dari
result
```

```
            e1, ok1 := Elements[ing1]
            e2, ok2 := Elements[ing2]
            if !ok1 || !ok2 {
                continue
            }

            if e1.Tier >= elem.Tier || e2.Tier >= elem.Tier {
                continue
            }

            // Buat node ingredient 1
            if _, exists := nodeMap[ing1]; !exists {
                step++
                nodeMap[ing1] = &Node{ Result: ing1, Step: step }
                queue = append(queue, nodeMap[ing1])
            }

            // Buat node ingredient 2
            if _, exists := nodeMap[ing2]; !exists {
                step++
                nodeMap[ing2] = &Node{ Result: ing2, Step: step }
                queue = append(queue, nodeMap[ing2])
            }

            current.Ingredient1 = nodeMap[ing1]
            current.Ingredient2 = nodeMap[ing2]
            break // resep valid pertama
        }
    }
    return root;
}
func BFSShortestNode(target string, Elements map[string]Element,
resultChan chan Message) {
    visitedNode := 0
    start := time.Now().UnixNano()
    root := BFSShortestNodeImpl(target, Elements, &visitedNode)
    resultChan <- Message{
        RecipePath: FlattenTreeToRecipePaths(root),
        NodesVisited: visitedNode,
```

```
        Duration: float32(time.Now().UnixNano() - start) /
1000000,
    }
}

func FlattenTreeToRecipePaths(root *Node) []RecipePath {
    var nodes []*Node
    var recipePaths []RecipePath

    var iter func(node *Node)
    iter = func(node *Node) {
        if node == nil {
            return
        }
        iter(node.Ingredient1)
        iter(node.Ingredient2)
        nodes = append(nodes, node)
    }
    iter(root)

    sort.Slice(nodes, func(i, j int) bool {
        return nodes[i].Step < nodes[j].Step
    })

    for _, node := range nodes {
        if node.Ingredient1 == nil && node.Ingredient2 == nil {
            // Ini BaseElement (daun)
            recipePaths = append(recipePaths, RecipePath{
                Ingredient1: "",
                Ingredient2: "",
                Result: node.Result,
            })
        } else {
            // Ini kombinasi normal
            recipePaths = append(recipePaths, RecipePath{
                Ingredient1: node.Ingredient1.Result,
                Ingredient2: node.Ingredient2.Result,
                Result:      node.Result,
            })
```

```
        }
    }
    return recipePaths
}
```

4.3. Implementasi Algoritma DFS

Implementasi algoritma DFS sesuai dengan teori, yaitu menjelajahi salah satu resep hingga menyentuh elemen dasar lalu *backtracking* ke elemen atas hingga semua simpul dijelajahi.

dfs.go

```
package utils

import (
    // "fmt"
    "maps"
    "sync"
    "time"
)

func DFSHelper(recipeMap RecipeMap, recipesEl RecipeElement,
targetElement string, visited map[string]bool, nodesVisited* int,
currentDepth int) []RecipePath {
    // Base case
    *nodesVisited++
    target, targetExists := recipesEl[targetElement]
    if !targetExists {
        return []RecipePath{}
    }
    if visited[targetElement] || BaseElement[targetElement] {
        return []RecipePath{{ Ingredient1: "", Ingredient2: "",
Result: targetElement }}
    }

    visited[targetElement] = true
    result := []RecipePath{}

    // Find recipe to create this element
```

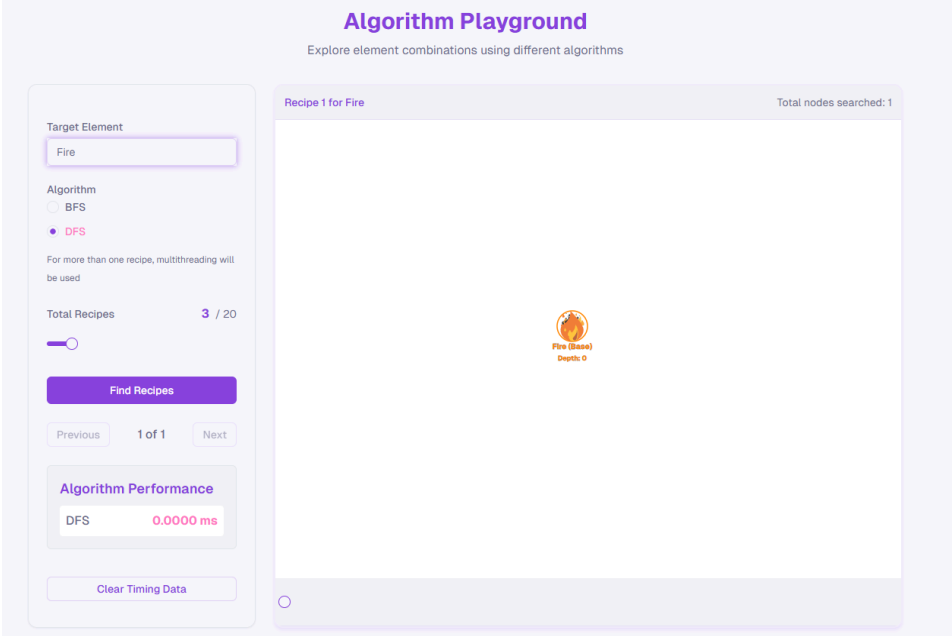
```
    for _, recipe := range target.Recipes {
        ing1Name := recipe[0]
        ing2Name := recipe[1]
        // fmt.Printf("Combination: %s, Result: %s\n",
combination, targetElement)
        // fmt.Printf("Ingredients: %s, %s\n", ing1, ing2)

        // Skip yang ga ada di resepnya
        if ing1Name == "Time" || ing2Name == "Time" {
            continue
        }
        ing1, ing1Exists := recipesEl[ing1Name]
        ing2, ing2Exists := recipesEl[ing2Name]
        if !ing1Exists || !ing2Exists {
            continue
        }
        if ing1.Tier >= target.Tier || ing2.Tier >= target.Tier {
            continue
        }

        var wg sync.WaitGroup
        wg.Add(2)
        ing1Channel := make(chan []RecipePath)
        ing2Channel := make(chan []RecipePath)
        go func() {
            defer wg.Done()
            visited1 := make(map[string]bool)
            maps.Copy(visited1, visited)
            subPath1 := DFSHelper(recipeMap, recipesEl, ing1Name,
visited1, nodesVisited, currentDepth + 1)
            ing1Channel <- subPath1

        }()
        go func() {
            defer wg.Done()
            visited2 := make(map[string]bool)
            maps.Copy(visited2, visited)
            subPath2 := DFSHelper(recipeMap, recipesEl, ing2Name,
visited2, nodesVisited, currentDepth + 1)
```

```
            ing2Channel <- subPath2

        }()
        subPath1 := <-ing1Channel
        subPath2 := <-ing2Channel
        wg.Wait()

        result = append(result, RecipePath{ Ingredient1: ing1Name,
Ingredient2: ing2Name, Result: targetElement })
        result = append(result, subPath1...)
        result = append(result, subPath2...)
        return result
    }
    return result
}

func DFS(recipeMap RecipeMap, recipesEl RecipeElement,
targetElement Element, resultChan chan Message) {
    visited := make(map[string]bool)
    nodesVisited := 0
    start := time.Now().UnixNano()
    if _, exists := recipesEl[targetElement.Name]; !exists {
        resultChan <- Message{
            RecipePath: []RecipePath{},
            NodesVisited: 0,
            Duration: float32(time.Now().UnixNano() - start) /
1000000,
        }
        return
    }
    recipePath := DFSHelper(recipeMap, recipesEl,
targetElement.Name, visited, &nodesVisited, 0)
    resultChan <- Message{
        RecipePath: recipePath,
        NodesVisited: nodesVisited,
        Duration: float32(time.Now().UnixNano() - start) /
1000000,
    }
```

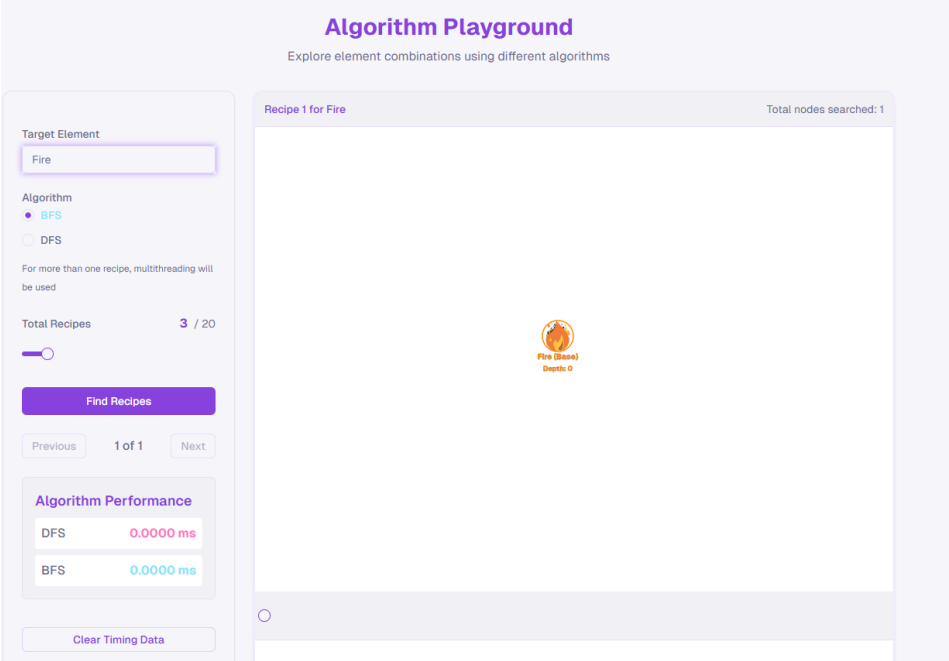
4.4. Eksperimen 1

Uji coba pencarian elemen dasar (*Tier* 0)

4.3.1. Depth First Search



4.3.2. Breadth First Search



Pada Eksperimen 1, kami menguji untuk pencarian Elemen Dasar, pada kedua percobaan tersebut hanya menghasilkan 1 elemen karena elemen yang dicari tidak perlu untuk dicari lagi turunannya.

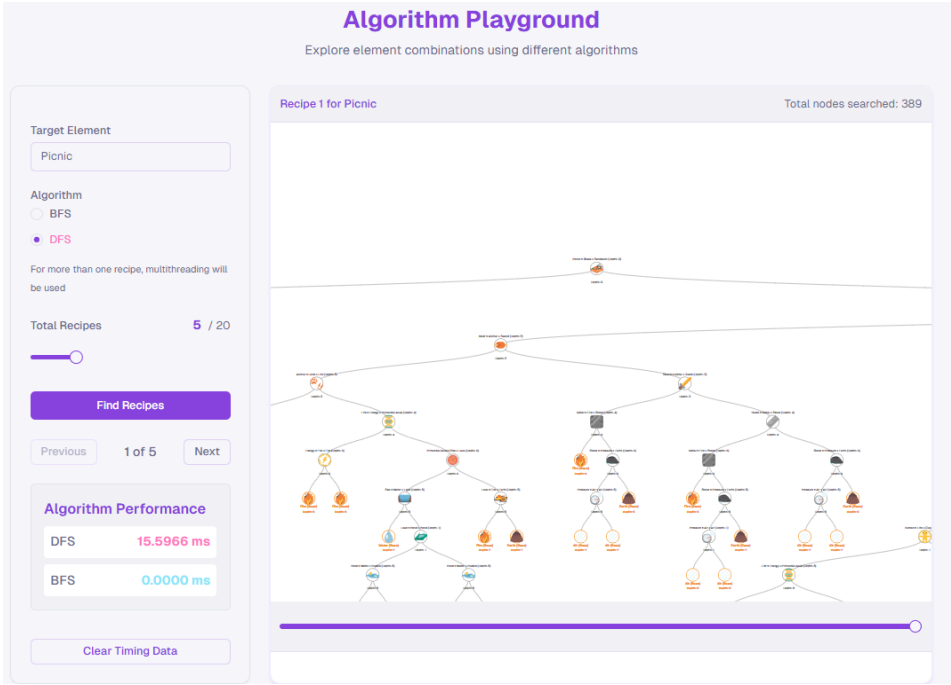
Untuk DFS, Elemen Dasar akan menjadi basis rekursi algoritma sehingga DFS akan berhenti ketika semua leaf menjadi elemen dasar

Untuk BFS, ketika menemukan elemen dasar, maka tidak akan diproses lebih lanjut. BFS akan berhenti melakukan pencarian ketika semua elemen di Queue kosong.

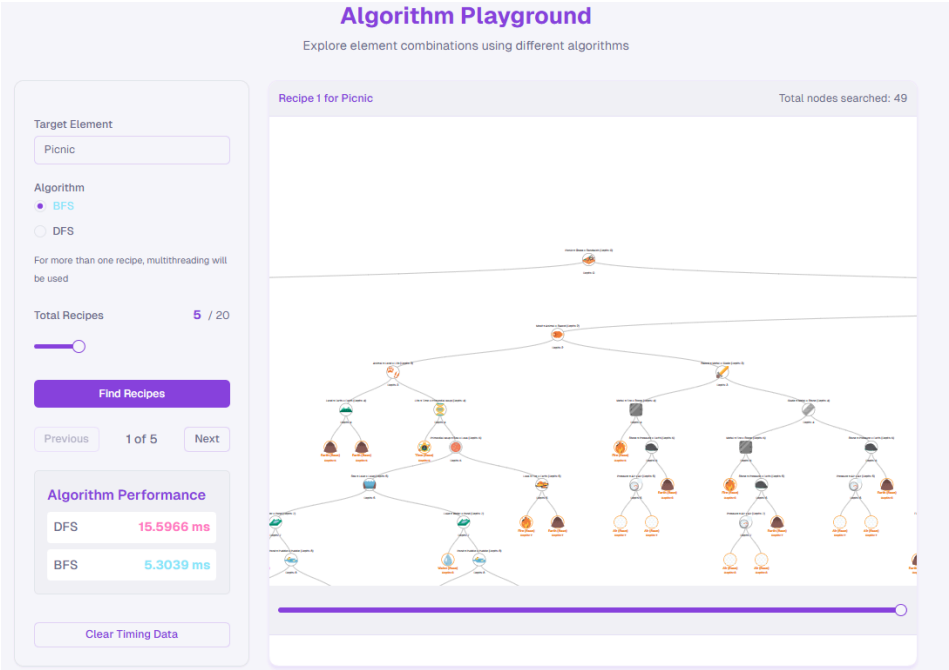
4.5. Eksperimen 2

Uji coba pencarian elemen dengan *tier* tertinggi (*Tier* 15)

4.3.1. Depth First Search



4.3.2. Breadth First Search

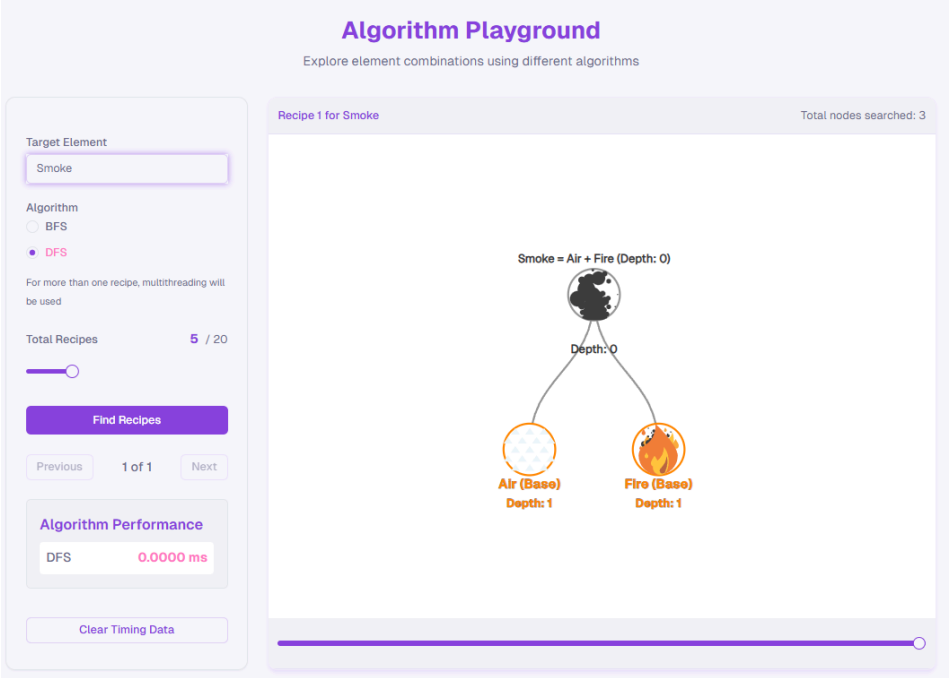


Pada Eksperimen 2, kami menguji untuk pencarian Elemen pada Tier 15,. Pada Tier ini, kami ingin memastikan apakah DFS dan BFS mampu untuk mencapai semua leaf hingga ke elemen dasar. Pada kasus ini, Target Elemen (Picnic) mampu dicari resepnya baik menggunakan DFS atau BFS.

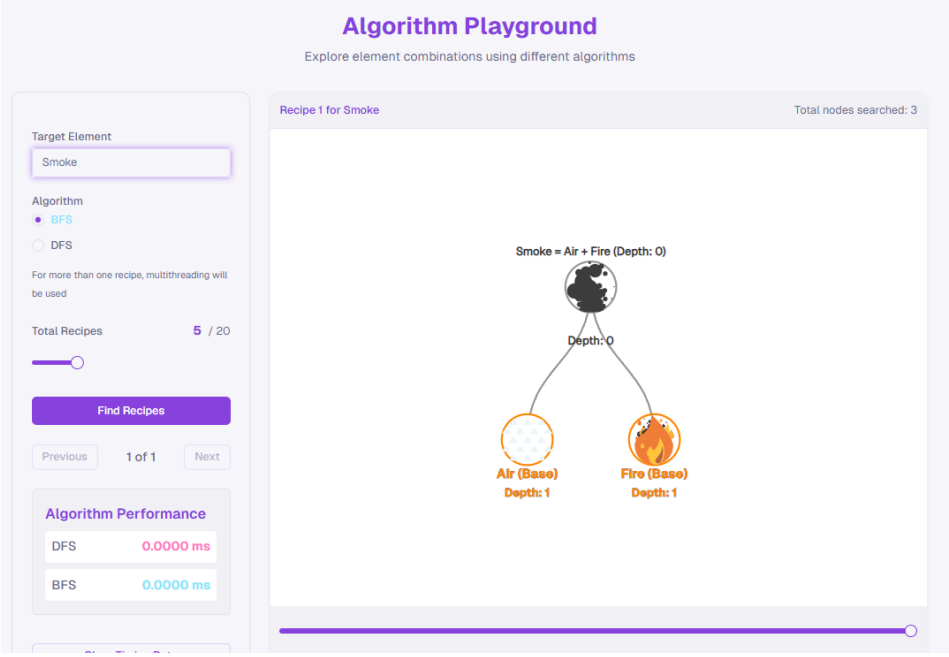
4.6. Eksperimen 3

Uji coba kasus samping (*edge cases)* adanya elemen waktu dalam resep

4.3.1. Depth First Search



4.3.2. Breadth First Search



Pada Eksperimen 3, kami menguji untuk pencarian Elemen Smoke, untuk mengetes uji resep yang rekursif.

BAB V ANALISIS, KESIMPULAN, DAN SARAN

5.1. Analisis

5.1.1. Breadth First Search

Implementasi Breadth-First Search (BFS) untuk mencari beberapa jalur resep menuju elemen target. BFS dimulai dari elemen target sebagai akar, kemudian mundur ke bahan-bahan penyusunnya hingga mencapai elemen dasar seperti *Air*, *Water*, *Earth*, atau *Fire*.

Setiap node dalam BFS mewakili suatu elemen (hasil dari suatu resep), dan anak-anaknya adalah dua node resep yang menyusunnya. Proses ini berulang hingga didapatkan sejumlah *N* jalur berbeda yang valid. Sehingga kompleksitas waktu dari algoritma ini adalah $O(b^d)$ dimana b adalah banyak kemungkinan kombinasi dan d adalah banyak cabang yang dibentuk dengan *best case*-nya adalah $O(1)$.

5.1.2. Depth First Search

Pendekatan Depth-First Search (DFS) untuk mencari berbagai variasi jalur resep menuju elemen target. DFS melakukan traversal secara mendalam—dari elemen target ke bahan, lalu terus ke bahan-bahan penyusun berikutnya—hingga mencapai elemen dasar.

Pada setiap langkah, DFS memilih satu kombinasi bahan secara deterministik berdasarkan seed (variasi urutan), lalu mengeksplorasi masing-masing bahan tersebut secara rekursif sehingga kompleksitas waktu dari algoritma ini adalah $O(b^d)$ dimana b adalah banyak kemungkinan kombinasi dan d adalah banyak cabang yang dibentuk dengan *best case*-nya adalah $O(1)$.

5.2. Kesimpulan

Dalam analisis algoritma pencarian resep pada game *Little Alchemy 2*, algoritma BFS dan DFS memiliki pendekatan dan keunggulan yang berbeda. Algoritma BFS(Breadth-First Search) bekerja dengan mengeksplorasi jalur dari target menuju bahan dasar secara menyeluruh pada setiap tingkat kedalaman sebelum melanjutkan ke tingkat berikutnya. Hal ini membuat BFS sangat efektif dalam menemukan jalur yang paling

pendek atau paling dangkal menuju elemen dasar. Namun, karena BFS menyimpan semua *node* yang sedang dan akan dieksplorasi dalam antrian, konsumsi memorinya cenderung tinggi, terutama jika jumlah kombinasi(branching factor) besar dan pohonnya dalam(deep).

Sementara itu, algoritma DFS(Depth-First Search) fokus pada penelusuran mendalam terlebih dahulu hingga mencapai elemen dasar, sebelum kembali dan mengeksplorasi jalur lain. DFS lebih efisien dalam penggunaan memori karena hanya menyimpan satu jalur eksplorasi pada satu waktu. Pendekatan ini cocok digunakan untuk mengeksplorasi berbagai variasi pohon resep yang mungkin ada, apalagi jika ingin menghasilkan hasil berbeda berdasarkan seed atau urutan kombinasi bahan. Akan tetapi, DFS tidak menjamin bahwa jalur yang ditemukan adalah yang paling optimal dari sisi kedalaman atau jumlah langkah. Oleh karena itu, pemanfaatan kedua algoritma ini secara bersamaan yaitu BFS untuk efisiensi dan optimalisasi, DFS untuk variasi dan eksplorasi yang dalam.

5.3. Saran

Untuk pengembangan lebih lanjut, sistem pencarian resep ini dapat ditingkatkan dengan menambahkan *heuristic search* yang memungkinkan pemanfaatan informasi tambahan seperti tingkat (*tier*) atau kompleksitas elemen untuk memandu pencarian ke arah yang lebih menjanjikan. Selain itu, integrasi sistem *caching* atau *memoization* untuk menyimpan hasil pencarian sub-resep yang sudah pernah dihitung juga bisa mengurangi redudansi pemrosesan, mempercepat pencarian, dan mengurangi konsumsi sumber daya.

Dari sisi pengalaman pengguna, pengembangan antarmuka visual yang mampu menampilkan pohon resep secara interaktif akan sangat membantu pengguna dalam memahami proses kombinasi yang terjadi. Pengguna juga dapat diberi opsi untuk menentukan kriteria pencarian, seperti memilih jalur tercepat, paling kompleks, atau paling kreatif. Akhirnya, untuk meningkatkan skalabilitas sistem, dukungan penuh terhadap multi-threading dan distribusi tugas ke layanan berbasis cloud bisa menjadi arah pengembangan jangka panjang agar sistem mampu melayani banyak permintaan sekaligus secara efisien.

BAB VI LAMPIRAN

6.1. Tautan

Link Github: Tubes2_Nuggets

Link Deployment: Tubes2_Nuggets

Link Dokumen: Tubes2_Nuggets

6.2. Tabel Evaluasi Fungsionalitas

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.	✅	