

**TUGAS KECIL 3**  
**IF2211 STRATEGI ALGORITMA**  
**Pemanfaatan Algoritma UCS, GBFS, A\* dalam Permainan**  
**Word Ladder**



**DISUSUN OLEH:**

**Jason Fernando**

**13522156**

**PROGRAM STUDI TEKNIK INFORMATIKA**  
**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**  
**INSTITUT TEKNOLOGI BANDUNG**  
**2024**

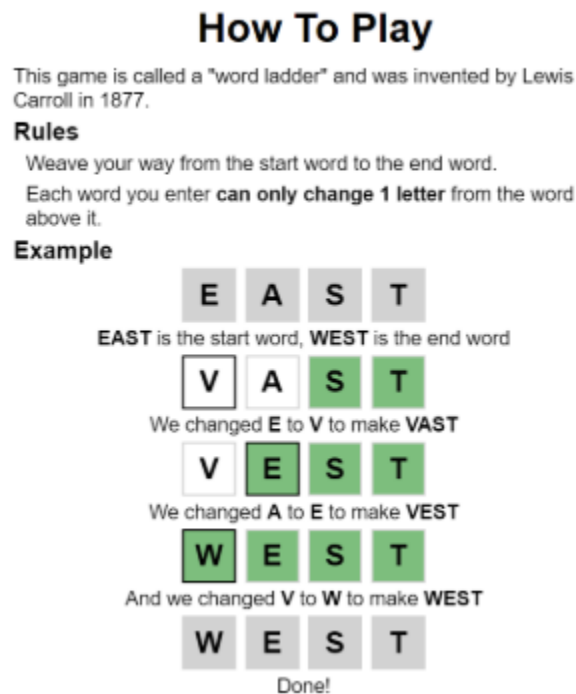
# DAFTAR ISI

<b>2.1 UCS (Uniform Cost Search).....</b>	<b>4</b>
<b>2.2 GBFS (Greedy Best First Search).....</b>	<b>5</b>
<b>2.3 A*.....</b>	<b>5</b>
<b>2.4 Website Development.....</b>	<b>5</b>
2.4.1 Backend Development.....	5
2.4.1 Frontend Development.....	6
<b>3.1 Definisi dari <math>f(n)</math> dan <math>g(n)</math>.....</b>	<b>7</b>
3.1.1 UCS (Uniform Cost Search).....	7
3.1.2 GBFS (Greedy Best First Search).....	7
3.1.3 A*.....	8
<b>3.2 Heuristik pada Algoritma A*.....</b>	<b>8</b>
<b>3.3 Algoritma UCS dan BFS.....</b>	<b>8</b>
<b>3.4 Algoritma UCS dan A*.....</b>	<b>9</b>
<b>3.5 Apakah algoritma BFS menjamin solusi optimal.....</b>	<b>9</b>
<b>4.1 Source Code Program.....</b>	<b>10</b>
4.1.1 ucs.java.....	10
4.1.1 gbfs.java.....	13
4.1.1 astar.java.....	16
<b>4.2 Implementasi Bonus.....</b>	<b>24</b>
<b>4.3 Pengujian.....</b>	<b>26</b>
4.3.1 UCS (Uniform Cost Search).....	26
4.3.2 GBFS (Greedy Best First Search).....	27
4.3.3 A*.....	29
<b>4.4 Hasil Analisis.....</b>	<b>30</b>
<b>5.1 Kesimpulan.....</b>	<b>33</b>

# BAB I

## DESKRIPSI MASALAH

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan. Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder



**Gambar 1.1** Contoh Permainan pada Web Weaver

## **BAB II**

### **LANDASAN TEORI**

#### **2.1 UCS (Uniform Cost Search)**

UCS (Uniform Cost Search) menggunakan struktur data graf, di mana graf merupakan representasi visual dari objek dan hubungan di antara mereka. Graf terdiri dari simpul (node) yang mewakili objek, dan sisi (edge) yang mewakili hubungan antara objek tersebut. Setiap sisi memiliki bobot (weight) yang menunjukkan biaya atau jarak antara dua simpul yang terhubung. Bobot dapat berupa angka real atau integer, yang bisa mewakili informasi seperti jarak fisik, waktu tempuh, atau biaya. Untuk memulai pencarian jalur terpendek, UCS memerlukan node awal (start node) dan node tujuan (goal node). Node awal adalah simpul dari mana pencarian dimulai, sementara node tujuan adalah simpul yang ingin dicapai.

UCS menggunakan explored set untuk melacak simpul-simpul yang telah dieksplorasi dan yang belum. Selama prosesnya, UCS juga menghitung biaya akumulatif (cost-to-come) dari jalur yang telah dieksplorasi. Biaya akumulatif ini digunakan untuk membandingkan jalur-jalur yang berbeda dan memilih jalur dengan biaya terendah. Biaya tersebut kemudian dimasukkan ke dalam priority queue untuk mengurutkan simpul-simpul berdasarkan biaya akumulatif jalur yang telah dieksplorasi. Simpul dengan biaya akumulatif yang lebih rendah akan dieksplorasi lebih dahulu karena diberi prioritas lebih tinggi.

#### **2.2 GBFS (Greedy Best First Search)**

Greedy Best-First Search (GBFS) adalah algoritma pencarian yang menggunakan pendekatan heuristik untuk memilih simpul berikutnya dalam pencarian. Algoritma ini memprioritaskan simpul yang memiliki nilai heuristik paling rendah, yang didasarkan pada estimasi jarak langsung ke simpul tujuan. Langkah-langkah dalam GBFS termasuk inisialisasi sebuah priority queue dengan simpul awal, kemudian selama priority queue tidak kosong, simpul dengan nilai heuristik paling rendah diambil. Jika simpul yang diambil adalah simpul tujuan, pencarian selesai.

Jika tidak, simpul tersebut ditambahkan ke explored set dan tetangga-tetangganya yang belum dieksplorasi dievaluasi. Setiap tetangga yang belum dieksplorasi akan dihitung nilai heuristiknya berdasarkan estimasi jarak langsung ke simpul tujuan, dan kemudian ditambahkan ke priority queue. GBFS tidak mempertimbangkan biaya akumulatif dari jalur yang telah dieksplorasi. Meskipun cenderung cepat, GBFS tidak selalu menjamin menemukan jalur terpendek, karena sifatnya yang greedy. Keberhasilan GBFS sangat tergantung pada pemilihan heuristik yang akurat untuk estimasi jarak ke simpul tujuan.

#### **2.3 A\***

A\* adalah algoritma pencarian yang memadukan strategi pencarian terinformasi dan estimasi heuristik untuk menemukan jalur terpendek dalam graf. Dengan memilih simpul berikutnya berdasarkan biaya total yang terkecil, yang merupakan gabungan dari biaya sejauh ini dan estimasi biaya sisa, A\* mampu menavigasi ruang pencarian dengan efisien. Langkah-langkahnya termasuk inisialisasi priority queue dengan simpul awal dan menghitung estimasi biaya sisa ke simpul tujuan. Selama priority queue memiliki simpul yang belum dieksplorasi, A\* mengevaluasi tetangga-tetangga dari simpul yang dipilih, menambahkan mereka ke dalam queue dengan memperbarui biaya total. Keunggulan utamanya terletak

pada kemampuannya dalam memilih jalur yang efisien dengan memanfaatkan informasi heuristik. Namun, keberhasilannya sangat tergantung pada keakuratan estimasi biaya sisa.

## **2.4 Website Development**

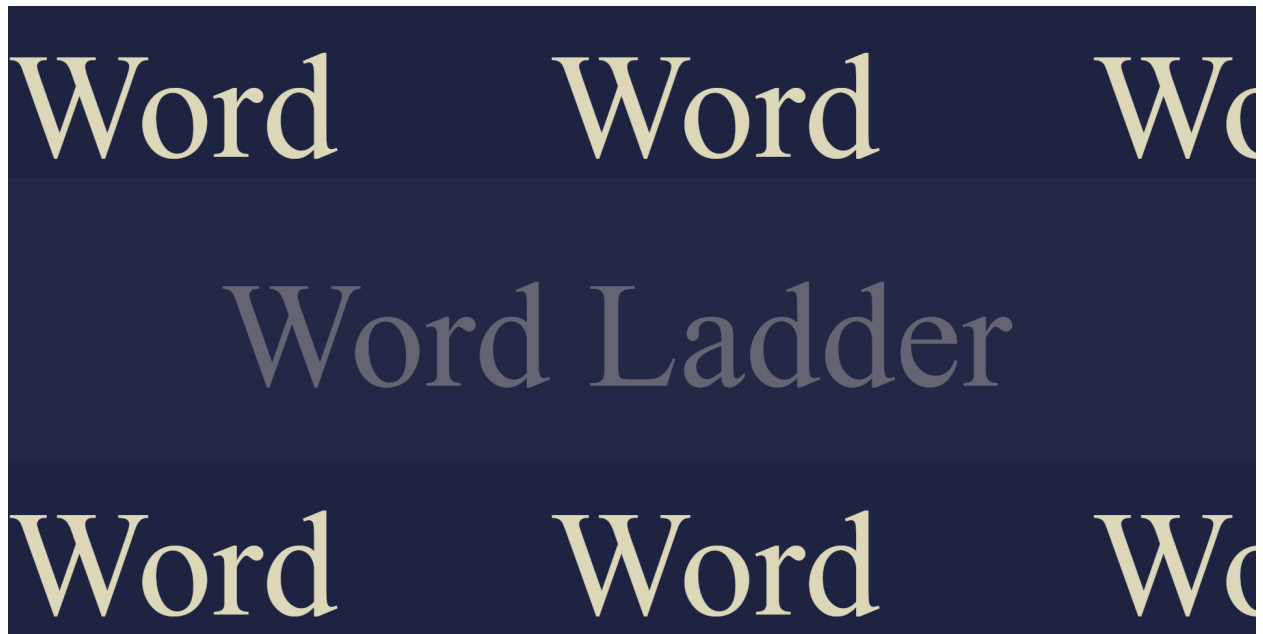
Luaran/output dari program ini adalah sebuah website, dalam pengembangan website ini kami menggunakan beberapa dependency dan juga framework. Pengembangan website dibagi menjadi dua yaitu frontend dan backend. Pada proses implementasi source code dari backend dan juga frontend dijalankan secara terpisah. Pengembangan backend menggunakan bahasa pemrograman Java sedangkan pengembangan frontend menggunakan bahasa pemrograman javascript.

### **2.4.1 Backend Development**

Java adalah bahasa pemrograman yang populer dan kuat yang digunakan dalam pengembangan berbagai jenis perangkat lunak, mulai dari aplikasi desktop hingga perangkat lunak berbasis web dan perangkat seluler. Salah satu fitur paling mencolok dari Java adalah kemampuannya untuk menjadi platform independen. Ini berarti bahwa program Java dapat dijalankan di berbagai sistem operasi tanpa perlu mengubah kode programnya, berkat penggunaan konsep "Write Once, Run Anywhere" (WORA) dan Java Virtual Machine (JVM). Java juga adalah bahasa pemrograman yang berbasis objek sepenuhnya (OOP), yang memungkinkan pembuatan kode yang modular, mudah dipahami, dan mudah di-maintain. Sintaksis Java yang mirip dengan bahasa pemrograman C dan C++ membuatnya mudah dipelajari bagi programmer yang sudah memiliki pengalaman dengan bahasa-bahasa tersebut. Java juga memiliki fitur keamanan bawaan yang kuat, termasuk mekanisme keamanan yang membatasi akses ke sumber daya sistem dan mencegah tindakan berbahaya dari program. Dengan komunitas pengembang yang besar dan aktif, serta banyaknya framework dan library yang tersedia, Java tetap menjadi salah satu bahasa pemrograman paling populer dan digunakan secara luas di berbagai bidang pengembangan perangkat lunak.

### **2.4.1 Frontend Development**

JavaScript adalah bahasa pemrograman yang digunakan untuk membuat situs web interaktif dengan menambahkan efek visual, interaktivitas, dan manipulasi konten halaman secara dinamis. Sementara itu, React adalah sebuah pustaka JavaScript yang dikembangkan oleh Facebook untuk membangun antarmuka pengguna (UI) yang interaktif dan mudah dimengerti. Dengan konsep komponen yang memungkinkan pembuatan UI dari bagian-bagian kecil yang dapat digunakan kembali, React mempermudah pengembangan dan pemeliharaan kode. Keunggulan React termasuk kemampuannya dalam memperbarui tampilan secara efisien dengan menggunakan Virtual DOM, serta ekosistem yang kaya dengan pustaka dan alat bantu untuk memperluas fungsionalitas aplikasi web.



**Gambar 1.2** Tampilan home website

The image shows a dark blue form with a light blue border. The form is titled "Word Ladder" in a light blue serif font. It is divided into two main sections: "Input Here" and "Choose Algorithm". The "Input Here" section contains two text input fields, "First Text" and "Second Text", separated by the word "to". The "Choose Algorithm" section contains three radio buttons labeled "UCS", "GBFS", and "A\*", and a "Submit" button at the bottom.

**Gambar 1.3** Tampilan input website

# BAB III

## ANALISIS MASALAH

### 3.1 Definisi dari $f(n)$ dan $g(n)$

#### 3.1.1 UCS (Uniform Cost Search)

- $f(n)$  merupakan total cost dari node awal sampai node ke- $n$ .
- $g(n)$  merupakan cost dari root sampai node ke- $n$ .

Dalam Uniform Cost Search (UCS),  $f(n)$  merupakan biaya total dari node awal hingga node  $n$ , sedangkan  $g(n)$  adalah biaya dari node awal hingga node  $n$  tersebut. UCS menggunakan  $g(n)$  untuk menentukan urutan ekspansi node selanjutnya, dengan mempertimbangkan biaya aktual dari node awal hingga node saat ini.

#### 3.1.2 GBFS (Greedy Best First Search)

- $f(n)$  merupakan cost dari node ke- $n$  sampai ke node tujuan.
- $g(n)$  greedy best first search tidak menggunakannya.

Dalam Greedy Best First Search (GBFS),  $f(n)$  adalah biaya dari node  $n$  hingga tujuan atau nilai heuristik dari node  $n$ , sedangkan  $g(n)$  tidak digunakan. GBFS hanya mempertimbangkan nilai heuristik untuk memilih node yang akan diekspansi selanjutnya, tanpa memperhatikan biaya aktual dari node awal hingga node tersebut.

#### 3.1.3 A\*

- $f(n)$  merupakan total cost dari node awal sampai node ke- $n$  ditambah dengan nilai heuristik dari node ke- $n$  sampai node tujuan.
- $g(n)$  merupakan cost dari node awal sampai node ke- $n$ .

Dalam A\*,  $f(n)$  adalah biaya total dari node awal hingga node  $n$  ditambah nilai heuristik dari node  $n$  hingga tujuan, sedangkan  $g(n)$  adalah biaya dari node awal hingga node  $n$ . A\* menggunakan  $g(n)$  dan  $h(n)$  (nilai heuristik) untuk menentukan urutan ekspansi node berikutnya, dengan mempertimbangkan biaya aktual dari node awal hingga node saat ini serta nilai heuristik dari node tersebut.

### 3.2 Heuristik pada Algoritma A\*

Heuristik yang digunakan pada A\* dikatakan admissible jika nilai heuristiknya tidak pernah melebihi biaya sebenarnya dari node saat ini hingga tujuan. Dalam konteks ini, heuristik yang digunakan adalah jumlah karakter yang berbeda antara node saat ini dan node tujuan. Misalnya, dalam konteks pencarian jalur dalam kata-kata, heuristik tersebut menghitung berapa banyak karakter yang berbeda antara kata saat ini dengan kata tujuan.

Misalnya, jika kita sedang mencari jalur dari kata "cat" ke kata "dog", dan kita sedang berada di kata "bat", heuristik akan menghitung berapa banyak karakter yang berbeda antara "bat" dan "dog". Heuristik ini admissible karena tidak pernah memberikan estimasi yang lebih tinggi dari biaya sebenarnya yang diperlukan untuk mencapai tujuan, yang dalam hal ini adalah jumlah langkah yang diperlukan untuk mengubah "bat" menjadi "dog". Dengan menggunakan heuristik yang admissible, A\* dapat memberikan

jaminan bahwa solusi yang ditemukan adalah solusi optimal, karena tidak akan melewatkan solusi yang lebih baik akibat estimasi yang terlalu tinggi.

### 3.3 Algoritma UCS dan BFS

Algoritma UCS (Uniform Cost Search) dan BFS (Breadth-First Search) dapat menghasilkan path yang sama dalam kasus word ladder jika diimplementasikan dengan benar. Namun, kedua algoritma ini memiliki perbedaan fundamental dalam pendekatan pencariannya.

BFS melakukan eksplorasi pada setiap level secara berurutan, mulai dari node awal, kemudian mengeksplorasi semua node yang memiliki jarak satu langkah dari node awal, dan seterusnya, hingga menemukan node tujuan. Pendekatan ini menghasilkan path yang optimal dalam hal jumlah langkah, karena BFS menjamin menemukan solusi terdekat terlebih dahulu.

Di sisi lain, UCS mempertimbangkan biaya aktual dari node awal hingga node saat ini. UCS memprioritaskan node yang memiliki biaya lebih rendah, tanpa memperhatikan level atau jarak dari node awal. Dalam kasus word ladder, jika implementasi antrian prioritas pada UCS mempertimbangkan biaya dengan benar, algoritma ini juga dapat menghasilkan path yang optimal.

### 3.4 Algoritma UCS dan A\*

Perbedaan utama antara algoritma A\* dan UCS terletak pada penggunaan nilai heuristik. A\* menggunakan nilai heuristik yang menggambarkan perkiraan jarak atau biaya yang tersisa untuk mencapai tujuan dari setiap node, sedangkan UCS hanya mempertimbangkan biaya sebenarnya dari node awal hingga node saat ini. Dengan memanfaatkan nilai heuristik, A\* dapat membuat estimasi yang lebih baik tentang jalur terpendek menuju tujuan, karena menggabungkan biaya sebenarnya yang telah ditempuh dengan perkiraan biaya yang tersisa. Ini memungkinkan A\* untuk menghindari mengeksplorasi jalur yang tidak produktif, mempercepat pencarian, dan mengurangi jumlah node yang perlu dieksplorasi secara keseluruhan.

Dalam kasus word ladder, di mana kita mencari jalur terpendek antara dua kata dengan mengubah satu huruf pada setiap langkah, A\* dapat memilih node yang memiliki nilai  $f(n)$  (biaya total dari node awal hingga node saat ini ditambah nilai heuristik) yang lebih kecil. Ini menunjukkan jalur yang lebih potensial untuk mencapai tujuan dalam jumlah langkah yang lebih sedikit. Namun, efisiensi relatif dari A\* dibandingkan dengan UCS juga tergantung pada kualitas nilai heuristik yang digunakan. Jika nilai heuristik cukup akurat dan admissible (tidak pernah melebihi biaya sebenarnya), A\* cenderung lebih efisien karena mampu "melihat" lebih jauh ke depan dalam pencarian solusi terpendek.

### 3.5 Apakah algoritma BFS menjamin solusi optimal

Dalam konteks persoalan word ladder, di mana kita mencari jalur terpendek antara dua kata dengan mengubah satu huruf pada setiap langkah, algoritma GBFS mungkin tidak menghasilkan solusi optimal. Hal ini disebabkan oleh fakta bahwa GBFS hanya mempertimbangkan nilai heuristik dari setiap node tanpa memperhitungkan biaya sebenarnya (jumlah langkah yang dibutuhkan) untuk mencapai tujuan. Meskipun GBFS cenderung memilih node yang terlihat mendekati tujuan berdasarkan nilai



heuristiknya, tidak ada jaminan bahwa node tersebut merupakan pilihan terbaik untuk mencapai solusi terpendek.

## BAB IV

### IMPLEMENTASI DAN PENGUJIAN

#### 4.1 Source Code Program

##### 4.1.1 ucs.java

```
package backend.ucs;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.*;

public class ucs {
    static class WordNode {
        String word;
        int cost;
        WordNode parent;

        WordNode(String word, int cost, WordNode parent) {
            this.word = word;
            this.cost = cost;
            this.parent = parent;
        }
    }

    public static class Result {
        public List<String> ladder;
        public int totalNodesVisited;

        public Result(List<String> ladder, int totalNodesVisited) {
            this.ladder = ladder;
            this.totalNodesVisited = totalNodesVisited;
        }
    }

    public static String sendGetRequest(String urlString) throws
    Exception {
```

```

        URL url = new URL(urlString);
        HttpURLConnection conn = (HttpURLConnection)
url.openConnection();

        conn.setRequestMethod("GET");

        int responseCode = conn.getResponseCode();

        if (responseCode == HttpURLConnection.HTTP_OK) {
            BufferedReader in = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
            String inputLine;
            StringBuilder content = new StringBuilder();

            while ((inputLine = in.readLine()) != null) {
                content.append(inputLine);
            }

            in.close();
            conn.disconnect();

            return content.toString();
        } else {
            return "GET request not worked";
        }
    }

    public static Result findLadder(String startWord, String endWord,
Set<String> wordList) {
        int totalNodesVisited = 0;
        List<String> ladder = new ArrayList<>();
        Queue<WordNode> queue = new LinkedList<>();
        Set<String> visited = new HashSet<>();

        queue.offer(new WordNode(startWord, 0, null));
        visited.add(startWord);

        while (!queue.isEmpty()) {
            WordNode currentNode = queue.poll();

```

```

        String currentWord = currentNode.word;
        totalNodesVisited++;

        if (currentWord.equals(endWord)) {
            // Construct ladder path
            while (currentNode != null) {
                ladder.add(0, currentNode.word);
                currentNode = currentNode.parent;
            }
            return new Result(ladder, totalNodesVisited);
        }

        for (int i = 0; i < currentWord.length(); i++) {
            char[] charArray = currentWord.toCharArray();
            for (char c = 'a'; c <= 'z'; c++) {
                charArray[i] = c;
                String nextWord = new String(charArray);
                if (wordList.contains(nextWord) &&
!visited.contains(nextWord) && nextWord.length() == endWord.length()) {
                    queue.offer(new WordNode(nextWord,
currentNode.cost + 1, currentNode));
                    visited.add(nextWord);
                }
            }
        }

        return null;
    }
}

```

Class **WordNode** : Kelas ini mewakili sebuah node dalam pohon pencarian. Ini berisi kata, biaya, dan referensi ke node induknya.

Class **Result** : Kelas ini menyimpan hasil dari pencarian word ladder. Ini berisi tangga (urutan kata dari awal hingga akhir) dan jumlah total node yang dikunjungi selama pencarian.

Method **sendGetRequest** : Metode ini mengirimkan permintaan HTTP GET ke URL yang diberikan dan mengembalikan konten respons sebagai string. Ini tidak langsung terkait dengan masalah word ladder dan tampaknya merupakan metode utilitas yang mungkin digunakan untuk tujuan lain.

Method **findLadder** : Ini adalah metode utama yang mengimplementasikan algoritma word ladder. Ini mengambil tiga parameter: kata awal, kata akhir, dan himpunan kata-kata yang valid (kamus). Ini mengembalikan objek **Result**.

#### 4.1.1 gbfs.java

```
package backend.gbfs;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.*;

public class gbfs {
    static class WordNode {
        String word;
        int cost;
        WordNode parent;

        WordNode(String word, int cost, WordNode parent) {
            this.word = word;
            this.cost = cost;
            this.parent = parent;
        }
    }

    public static class Result {
        public List<String> ladder;
        public int totalNodesVisited;

        public Result(List<String> ladder, int totalNodesVisited) {
            this.ladder = ladder;
            this.totalNodesVisited = totalNodesVisited;
        }
    }

    public static String sendGetRequest(String urlString) throws
    Exception {
```

```

        URL url = new URL(urlString);
        HttpURLConnection conn = (HttpURLConnection)
url.openConnection();

        conn.setRequestMethod("GET");

        int responseCode = conn.getResponseCode();

        if (responseCode == HttpURLConnection.HTTP_OK) {
            BufferedReader in = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
            String inputLine;
            StringBuilder content = new StringBuilder();

            while ((inputLine = in.readLine()) != null) {
                content.append(inputLine);
            }

            in.close();
            conn.disconnect();

            return content.toString();
        } else {
            return "GET request not worked";
        }
    }

    public static Result findLadder(String startWord, String endWord,
Set<String> wordList) {
        int totalNodesVisited = 0;
        List<String> ladder = new ArrayList<>();
        PriorityQueue<WordNode> queue = new
PriorityQueue<>(Comparator.comparingInt(node -> heuristic(node.word,
endWord)));
        Set<String> visited = new HashSet<>();

        queue.offer(new WordNode(startWord, 0, null));
        visited.add(startWord);
    }

```

```

while (!queue.isEmpty()) {
    WordNode currentNode = queue.poll();
    String currentWord = currentNode.word;
    totalNodesVisited++;

    if (currentWord.equals(endWord)) {
        while (currentNode != null) {
            ladder.add(0, currentNode.word);
            currentNode = currentNode.parent;
        }
        return new Result(ladder, totalNodesVisited);
    }

    for (int i = 0; i < currentWord.length(); i++) {
        char[] charArray = currentWord.toCharArray();
        for (char c = 'a'; c <= 'z'; c++) {
            charArray[i] = c;
            String nextWord = new String(charArray);
            if (wordList.contains(nextWord) &&
!visited.contains(nextWord) && nextWord.length() == endWord.length()) {
                queue.offer(new WordNode(nextWord,
currentNode.cost + 1, currentNode));
                visited.add(nextWord);
            }
        }
    }

    return null;
}

static int heuristic(String word1, String word2) {
    int count = 0;
    for (int i = 0; i < word1.length(); i++) {
        if (word1.charAt(i) != word2.charAt(i)) {
            count++;
        }
    }
}

```

```

        return count;
    }
}

```

Class **WordNode** : Kelas ini mewakili sebuah node dalam pohon pencarian. Ini berisi kata, biaya, dan referensi ke node induknya.

Class **Result** : Kelas ini menyimpan hasil dari pencarian word ladder. Ini berisi tangga (urutan kata dari awal hingga akhir) dan jumlah total node yang dikunjungi selama pencarian.

Method **sendGetRequest** : Metode ini mengirimkan permintaan HTTP GET ke URL yang diberikan dan mengembalikan konten respons sebagai string. Ini tidak langsung terkait dengan masalah word ladder dan tampaknya merupakan metode utilitas yang mungkin digunakan untuk tujuan lain.

Method **findLadder** : Ini adalah metode utama yang mengimplementasikan algoritma word ladder. Ini mengambil tiga parameter: kata awal, kata akhir, dan himpunan kata-kata yang valid (kamus). Ini mengembalikan objek **Result**.

Method **heuristic** : Metode ini menghitung heuristik antara dua kata. Heuristik adalah perkiraan biaya yang diperlukan untuk mencapai tujuan (kata akhir) dari posisi saat ini (kata saat ini).

#### 4.1.1 astar.java

```

package backend.astar;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.*;

public class astar {
    static class WordNode {
        String word;
        int gCost;
        int hCost;
        WordNode parent;

        WordNode(String word, int gCost, int hCost, WordNode parent) {
            this.word = word;
            this.gCost = gCost;
            this.hCost = hCost;
            this.parent = parent;
        }
    }
}

```



```

        int getTotalCost() {
            return gCost + hCost;
        }
    }

    public static class Result {
        public List<String> ladder;
        public int totalNodesVisited;

        public Result(List<String> ladder, int totalNodesVisited) {
            this.ladder = ladder;
            this.totalNodesVisited = totalNodesVisited;
        }
    }

    public static String sendGetRequest(String urlString) throws
Exception {
        URL url = new URL(urlString);
        HttpURLConnection conn = (HttpURLConnection)
url.openConnection();

        conn.setRequestMethod("GET");

        int responseCode = conn.getResponseCode();

        if (responseCode == HttpURLConnection.HTTP_OK) {
            BufferedReader in = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
            String inputLine;
            StringBuilder content = new StringBuilder();

            while ((inputLine = in.readLine()) != null) {
                content.append(inputLine);
            }

            in.close();
            conn.disconnect();

            return content.toString();
        }
    }
}

```

```

    } else {
        return "GET request not worked";
    }
}

public static Result findLadder(String startWord, String endWord,
Set<String> wordList) {
    int totalNodesVisited = 0;
    List<String> ladder = new ArrayList<>();
    PriorityQueue<WordNode> queue = new
PriorityQueue<>(Comparator.comparingInt(node -> node.getTotalCost()));
    Map<String, Integer> gCostMap = new HashMap<>();
    Set<String> visited = new HashSet<>();

    WordNode startNode = new WordNode(startWord, 0,
heuristic(startWord, endWord), null);
    queue.offer(startNode);
    gCostMap.put(startWord, 0);

    while (!queue.isEmpty()) {
        WordNode currentNode = queue.poll();
        String currentWord = currentNode.word;
        totalNodesVisited++;

        if (currentWord.equals(endWord)) {
            while (currentNode != null) {
                ladder.add(0, currentNode.word);
                currentNode = currentNode.parent;
            }
            return new Result(ladder, totalNodesVisited);
        }

        visited.add(currentWord);

        for (int i = 0; i < currentWord.length(); i++) {
            char[] charArray = currentWord.toCharArray();
            for (char c = 'a'; c <= 'z'; c++) {
                charArray[i] = c;
                String nextWord = new String(charArray);

```

```

        if (wordList.contains(nextWord) &&
!visited.contains(nextWord) && nextWord.length() == endWord.length()) {
            int newGCost = currentNode.gCost + 1;
            if (!gCostMap.containsKey(nextWord) || newGCost
< gCostMap.get(nextWord)) {
                gCostMap.put(nextWord, newGCost);
                WordNode nextNode = new WordNode(nextWord,
newGCost, heuristic(nextWord, endWord), currentNode);
                queue.offer(nextNode);
            }
        }
    }
}

return null;
}

static int heuristic(String word1, String word2) {
    int count = 0;
    for (int i = 0; i < word1.length(); i++) {
        if (word1.charAt(i) != word2.charAt(i)) {
            count++;
        }
    }
    return count;
}
}

```

Class **WordNode** : Kelas ini merepresentasikan sebuah node dalam algoritma A\*. Setiap node memiliki atribut kata (word), biaya aktual (gCost), biaya perkiraan ke tujuan (hCost), dan referensi ke node induknya (parent). Atribut gCost menyimpan biaya aktual dari node awal ke node saat ini, sedangkan atribut hCost menyimpan nilai heuristik dari node saat ini ke node tujuan. Metode getTotalCost() digunakan untuk mengembalikan total biaya dari node (gCost + hCost).

Class **Result** : Kelas ini digunakan untuk menyimpan hasil dari pencarian tangga kata. Objek Result berisi list tangga kata (ladder) dan jumlah total node yang dikunjungi selama pencarian (totalNodesVisited).

Method **sendGetRequest** : Metode ini mirip dengan yang sebelumnya, digunakan untuk mengirimkan permintaan HTTP GET ke URL yang diberikan dan mengembalikan konten respons sebagai string.

Method **findLadder** : Metode utama yang mengimplementasikan algoritma A\* untuk mencari tangga kata terpendek dari kata awal ke kata akhir. Metode ini memiliki tiga parameter: kata awal, kata akhir, dan himpunan kata-kata yang valid (kamus). Metode ini mengembalikan objek **Result**.

Method **heuristic** : Metode ini menghitung heuristik antara dua kata. Heuristik adalah perkiraan biaya yang diperlukan untuk mencapai tujuan (kata akhir) dari posisi saat ini (kata saat ini).

#### 4.1.1 Main.java

```
package backend;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.URI;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

import com.sun.net.httpserver.HttpServer;

import backend.astar.astar;
import backend.gbfs.gbfs;
import backend.ucs.ucs;

public class Main {

    public static List<String> parseDictionary(String filePath) throws
    IOException {
        List<String> parsedList = new ArrayList<>();
        BufferedReader reader = new BufferedReader(new
        FileReader(filePath));
        String line;
        while ((line = reader.readLine()) != null) {
            parsedList.add(line.trim());
        }
    }
}
```

```

        reader.close();
        return parsedList;
    }

    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new
InetSocketAddress("0.0.0.0", 8000), 0);

        server.createContext("/findLadder", exchange -> {

exchange.getResponseHeaders().add("Access-Control-Allow-Origin", "*");

exchange.getResponseHeaders().add("Access-Control-Allow-Methods", "POST,
GET, OPTIONS");

exchange.getResponseHeaders().add("Access-Control-Allow-Headers",
"Content-Type");

            if ("OPTIONS".equals(exchange.getRequestMethod())) {
                exchange.sendResponseHeaders(204, -1);
                return;
            }

            if ("GET".equals(exchange.getRequestMethod())) {
                URI requestedUri = exchange.getRequestURI();
                String query = requestedUri.getRawQuery();
                Map<String, String> queryParams =
parseQueryParams(query);

                String startWord = queryParams.get("startWord");
                String endWord = queryParams.get("endWord");
                String algorithm = queryParams.get("algorithm");

                if (startWord == null || endWord == null || algorithm ==
null) {

                    exchange.sendResponseHeaders(400, -1);
                    return;
                }
            }
        }
    }
}

```

```

        Set<String> wordList;
        try {
            wordList = new
HashSet<>(parseDictionary("backend/dictionary/words_alpha.txt"));
        } catch (IOException e) {
            exchange.sendResponseHeaders(500, -1);
            return;
        }

        Object result = null;
        List<String> ladder = null;
        int totalNodesVisited = 0;

        switch (algorithm.toLowerCase()) {
            case "ucs":
                result = ucs.findLadder(startWord, endWord,
wordList);

                ladder = ((ucs.Result)result).ladder;
                totalNodesVisited =
((ucs.Result)result).totalNodesVisited;
                break;
            case "astar":
                result = astar.findLadder(startWord, endWord,
wordList);

                ladder = ((astar.Result)result).ladder;
                totalNodesVisited =
((astar.Result)result).totalNodesVisited;
                break;
            case "gbfs":
                result = gbfs.findLadder(startWord, endWord,
wordList);

                ladder = ((gbfs.Result)result).ladder;
                totalNodesVisited =
((gbfs.Result)result).totalNodesVisited;
                break;
            default:
                exchange.sendResponseHeaders(400, -1);
                return;
        }
    }

```

```

        StringBuilder jsonBuilder = new
StringBuilder("{\"path\":[");
        for (int i = 0; i < ladder.size(); i++) {

jsonBuilder.append("\"").append(ladder.get(i)).append("\"");
            if (i < ladder.size() - 1) {
                jsonBuilder.append(",");
            }
        }
        jsonBuilder.append("]");

jsonBuilder.append(",\"totalNodesVisited\":"+totalNodesVisited);
        jsonBuilder.append("}");
        String response = jsonBuilder.toString();

        exchange.getResponseHeaders().set("Content-Type",
"application/json");
        exchange.sendResponseHeaders(200, response.length());
        OutputStream os = exchange.getResponseBody();
        os.write(response.getBytes());
        os.close();
    } else {
        exchange.sendResponseHeaders(405, -1);
    }
});

server.start();
}

public static Map<String, String> parseQueryParams(String query) {
    Map<String, String> result = new HashMap<>();
    for (String param : query.split("&")) {
        String[] entry = param.split("=");
        if (entry.length > 1) {
            result.put(entry[0], entry[1]);
        } else {
            result.put(entry[0], "");
        }
    }
}

```

```
    }  
    return result;  
  }  
}
```

Method **parseDictionary** : Metode untuk mem-parsing kamus kata dari file teks, Mengembalikan daftar kata yang telah diparsing.

Method **parseQueryParams** : Metode untuk mem-parsing query string dari permintaan HTTP, Mengembalikan map yang berisi pasangan kunci-nilai dari parameter yang ditemukan dalam query string.

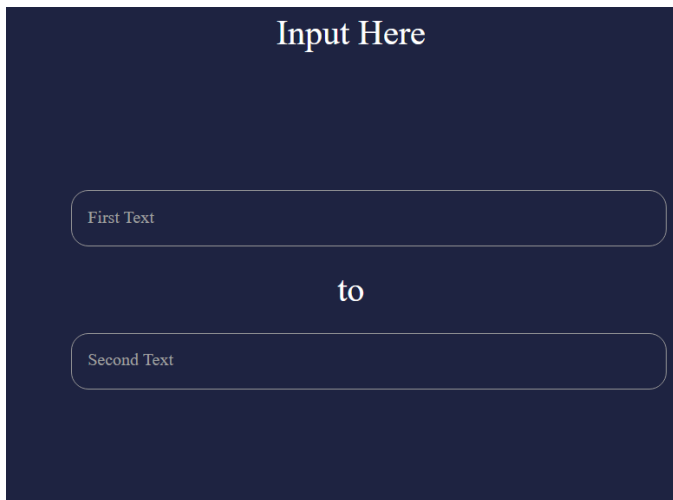
Method **main** : Metode utama untuk memanggil fungsi ucs, gbfs, astar serta menerima input value dari front-end dan juga mengembalikan hasil setelah algoritma.

## 4.2 Implementasi Bonus

Program bisa dijalankan dengan GUI (Graphical User Interface) menggunakan website dengan React.

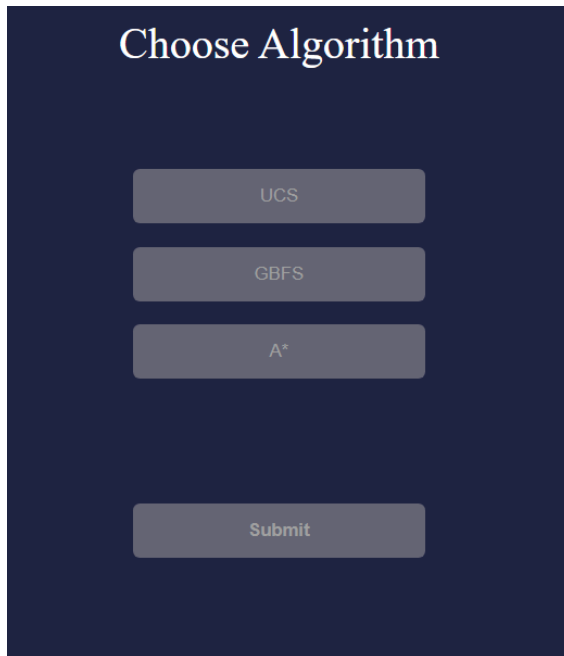
GUI memiliki komponen :

- Input kata pertama dan kedua yang hanya dapat menerima bahasa inggris valid

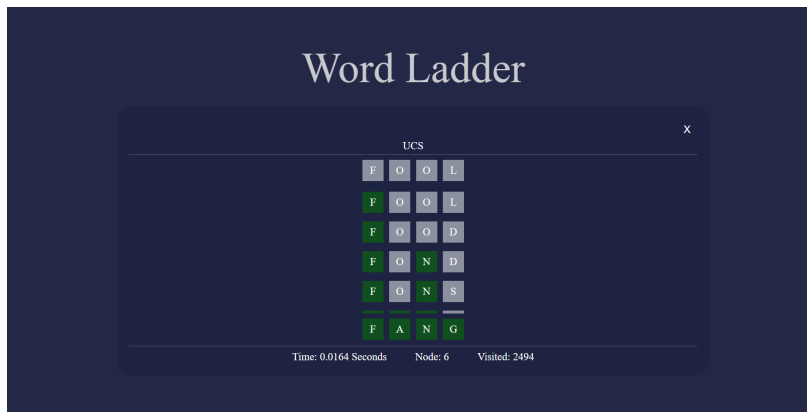




- Input Algoritma



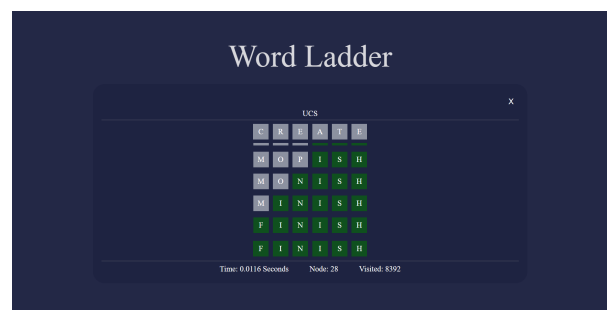
- Output



## 4.3 Pengujian

### 4.3.1 UCS (Uniform Cost Search)

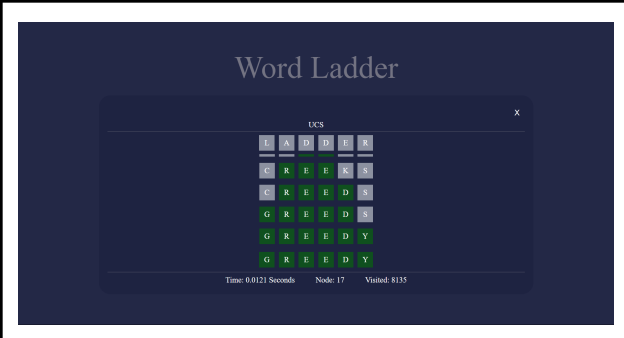
Start Word : create  
End Word : finish



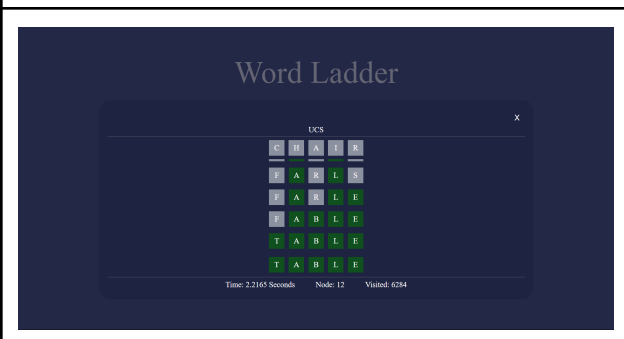
Start Word : dog  
End Word : cat



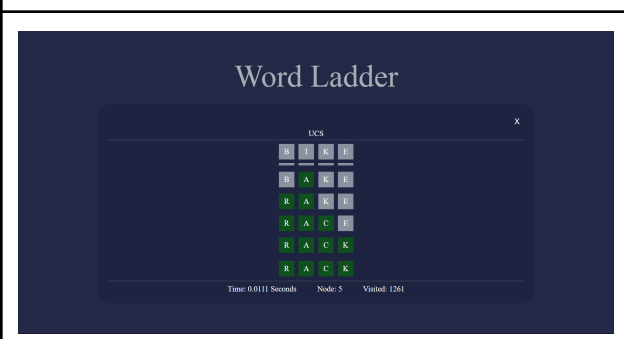
Start Word : ladder  
End Word : greedy



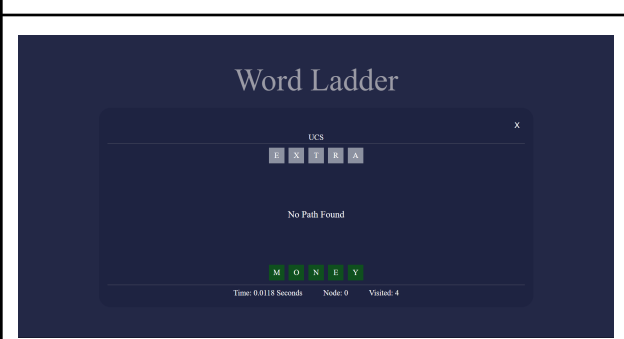
Start Word : chair  
End Word : table



Start Word : bike  
End Word : rack

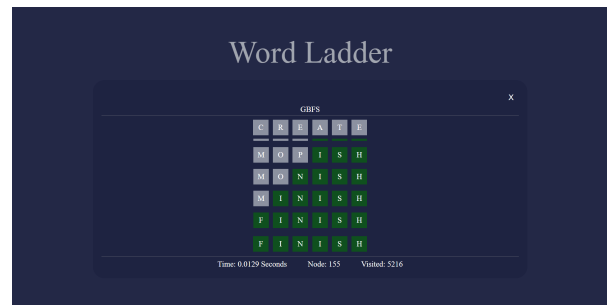


Start Word : extra  
End Word : money

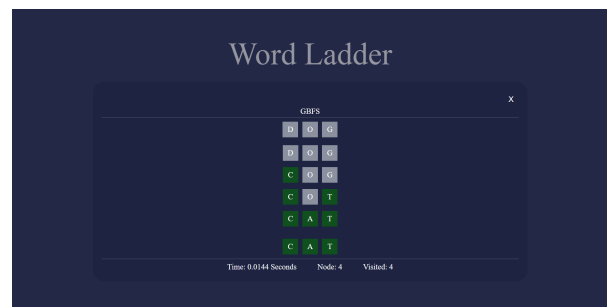


### 4.3.2 GBFS (Greedy Best First Search)

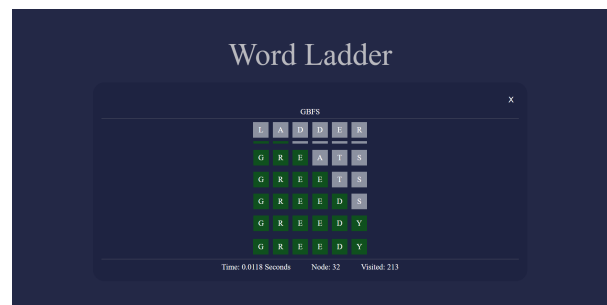
Start Word : create  
End Word : finish



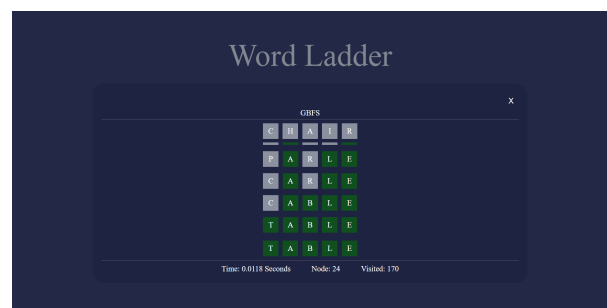
Start Word : dog  
End Word : cat



Start Word : ladder  
End Word : greedy




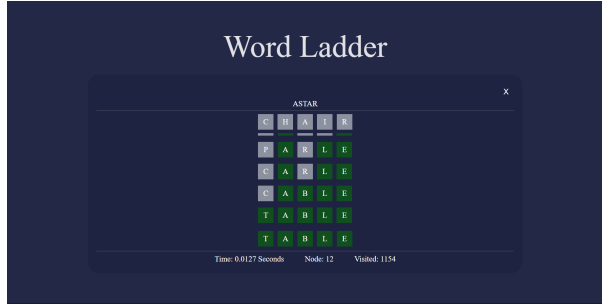
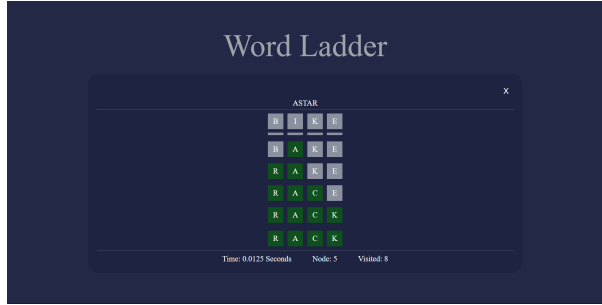

Start Word : chair  
End Word : table



<p>Start Word : bike End Word : rack</p>	
<p>Start Word : extra End Word : money</p>	

### 4.3.3 A\*

<p>Start Word : create End Word : finish</p>	
<p>Start Word : dog End Word : cat</p>	

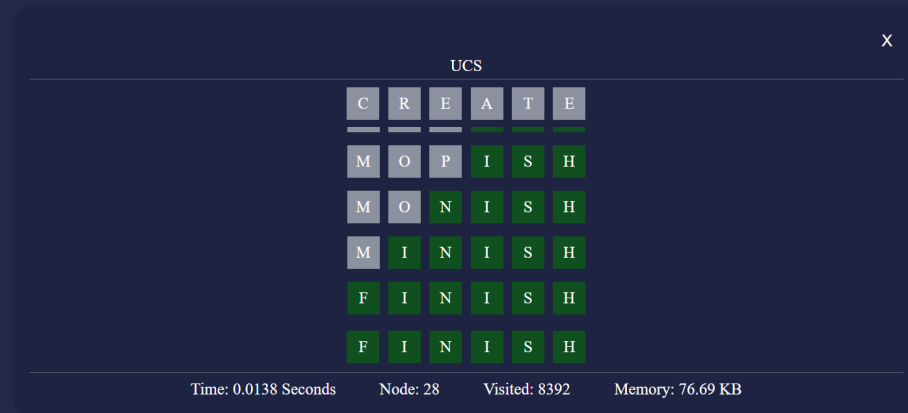
<p>Start Word : ladder End Word : greedy</p>	
<p>Start Word : chair End Word : table</p>	
<p>Start Word : bike End Word : rack</p>	
<p>Start Word : extra End Word : money</p>	

#### 4.4 Hasil Analisis

Berikut adalah hasil analisis dan bukti pendukung terkait analisis yang dilakukan menggunakan kata awal yaitu “create” dan kata akhir yaitu “finish”

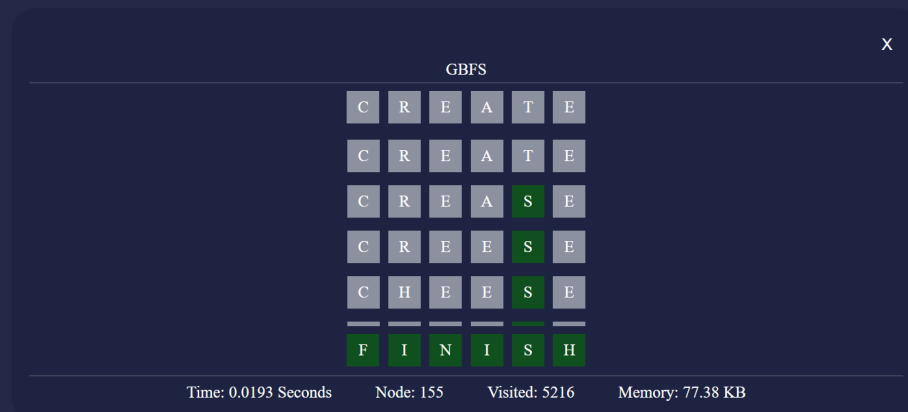
- UCS

# Word Ladder



- Optimalitas : Algoritma UCS optimal dikarenakan mengunjungi node dengan cost terendah terlebih dahulu
- Waktu Eksekusi : Algoritma UCS mempunyai waktu eksekusi yang bervariasi dilihat dari banyaknya node yang dikunjungi
- Memory : Algoritma UCS memerlukan tempat penyimpanan simpul yang belum dieksplorasi dan biaya setiap simpul. Namun, karena hanya menyimpan informasi biaya, memori yang dibutuhkan tidak terlalu besar dibanding memori yang dibutuhkan oleh algoritma gbfs dan a\*

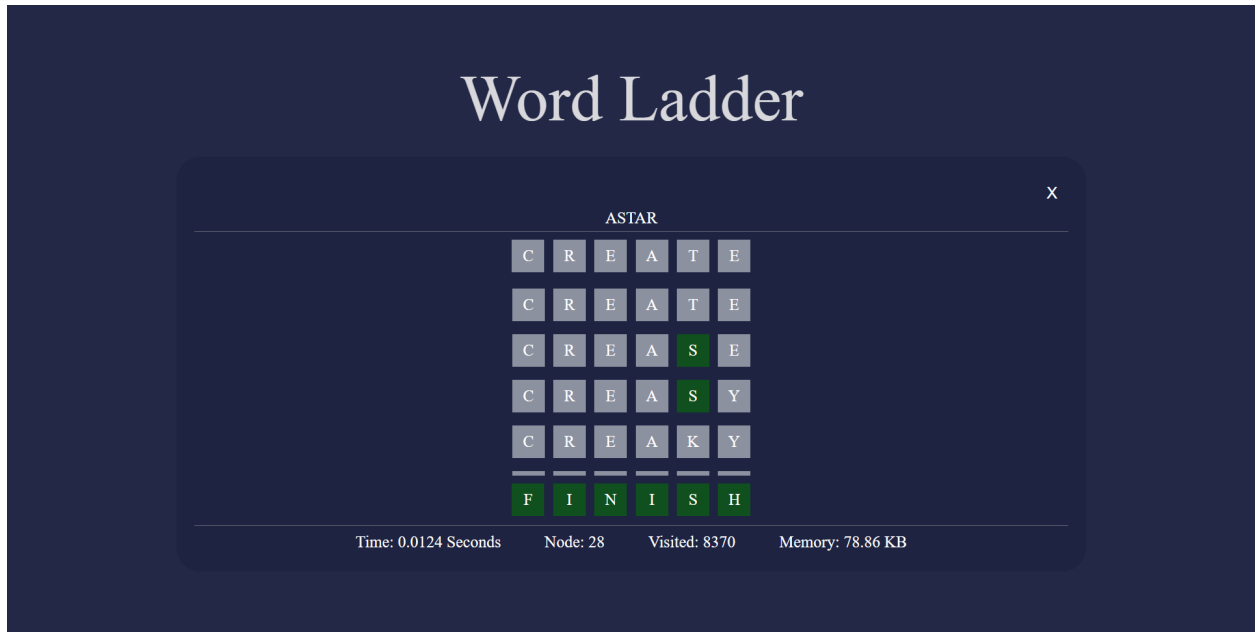
## • GBFS



- Optimalitas : Algoritma GBFS kurang optimal karena memperhatikan heuristik lokal

- Waktu Eksekusi : Algoritma GBFS dapat menemukan jalur dengan cepat jika heuristiknya mendekati solusi secara akurat. Sebaliknya jika tidak maka GBFS akan memakan waktu yang banyak.
- Memory : Algoritma GBFS membutuhkan memori untuk menyimpan simpul yang belum dieksplorasi dan nilai heuristik setiap simpul.

- A\*



- Optimalitas : Algoritma A\* optimal jika fungsi heuristiknya admissible dan konsisten.
- Waktu Eksekusi : Algoritma A\* memiliki waktu eksekusi yang bervariasi bergantung pada heuristik yang digunakan dan struktur graf yang dieksplorasi.
- Memory : Algoritma A\* memiliki memory yang terbanyak dikarenakan A\* membutuhkan memori untuk menyimpan simpul yang belum dieksplorasi, biaya setiap simpul, nilai heuristik, serta biaya total

# **BAB 5**

## **PENUTUP**

### **5.1 Kesimpulan**

Dalam mencari solusi yang optimal dalam permainan *word ladder* dapat diambil beberapa kesimpulan yaitu Algoritma UCS merupakan algoritma yang optimal serta memprioritaskan pengunjungan simpul berbiaya minimum, namun waktu eksekusinya mungkin lebih lama terutama jika terdapat banyak simpul dengan biaya rendah. Algoritma GBFS kurang menjamin optimalitas karena hanya mempertimbangkan heuristik lokal. Sedangkan Algoritma A\* merupakan algoritma yang optimal dengan waktu eksekusi yang relatif efisien, tetapi membutuhkan lebih banyak memori karena menyimpan informasi lebih lengkap dibandingkan algoritma UCS dan algoritma GBFS.



## DAFTAR PUSTAKA

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>

# LAMPIRAN

## LINK REPOSITORY

Link repository GitHub : [https://github.com/JasonFernandoo/Tucil3\\_13522156](https://github.com/JasonFernandoo/Tucil3_13522156)

Poin	Ya	Tidak
1. Program berhasil dijalankan.	✓	
2. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma UCS	✓	
3. Solusi yang diberikan pada algoritma UCS optimal	✓	
4. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma Greedy Best First Search	✓	
5. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma A*	✓	
6. Solusi yang diberikan pada algoritma A* optimal	✓	
7. [Bonus]: Program memiliki tampilan GUI	✓	