

LAPORAN TUGAS KECIL 03
IF2211 STRATEGI ALGORITMA

**Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS, Greedy
Best First Search, dan A***



Disusun oleh:

Satriadhikara Panji Yudhistira K-03 13522125

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

DAFTAR ISI

26

DAFTAR ISI	2
BAB 1	4
DESKRIPSI MASALAH	4
BAB 2	5
2.1. Algoritma Uniform Cost Search (UCS)	5
2.2. Algoritma Greedy Best First Search	5
2.3. Algoritma A*	5
BAB 3	6
3.1. Source Code Program	6
3.1.1. Algorithm.java	6
3.1.2. AStar.java	7
3.1.3. GreedyBestFirstSearch.java	8
3.1.4. UCS.java	9
3.1.5. Node.java	11
3.1.6. Dictionary.java	11
3.1.7. Api.java	13
3.1.8. InputWord.java	14
3.1.9. Response.java	14
3.1.10. BackendApplication.java	16
3.2. Implementasi Bonus	16
3.3. Definisi dari $f(n)$ dan $g(n)$	16
3.1.1. UCS	16
3.1.2. Greedy Best First Search	16
3.1.3. A*	17
3.4. Apakah heuristik yang digunakan pada algoritma A* admissible?	17
3.5. Pada kasus word ladder, apakah algoritma UCS sama dengan BFS?	17
3.6. Secara teoritis, apakah algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada kasus word ladder?	17
3.7. Secara teoritis, apakah algoritma Greedy Best First Search menjamin solusi optimal untuk persoalan word ladder?	18
BAB 4	19
4.1. Tampilan GUI	19
4.2. Uniform Cost Search	19
4.3. Greedy Best First Search	22
4.4. A*	25
BAB 5	30
5.1 Kesimpulan	30
5.2 Saran	30
DAFTAR PUSTAKA	31
LAMPIRAN	32

BAB 1

DESKRIPSI MASALAH

Dalam game Word Ladder, tujuan utama adalah untuk mengubah satu kata (kata awal) menjadi kata lain (kata akhir) dengan mengubah satu huruf pada suatu waktu. Setiap langkah harus menghasilkan kata yang valid dalam bahasa yang digunakan, biasanya bahasa Inggris. Masalah ini menjadi menarik dan menantang karena setiap perubahan harus mempertahankan struktur kata yang dapat dikenali dan benar secara leksikal.

Word Ladder adalah teka-teki atau permainan bahasa yang pertama kali diciptakan oleh Lewis Carroll pada tahun 1877. Dalam permainan ini, pemain diberikan dua kata yang memiliki panjang yang sama dan tugasnya adalah untuk menemukan jalur dari kata pertama ke kata kedua. Contohnya, untuk mengubah kata "COLD" menjadi "WARM", pemain mungkin membuat jalur sebagai berikut: COLD → CORD → CARD → WARD → WARM. Setiap langkah hanya melibatkan perubahan satu huruf, dan setiap kata hasil perubahan harus merupakan kata yang valid.

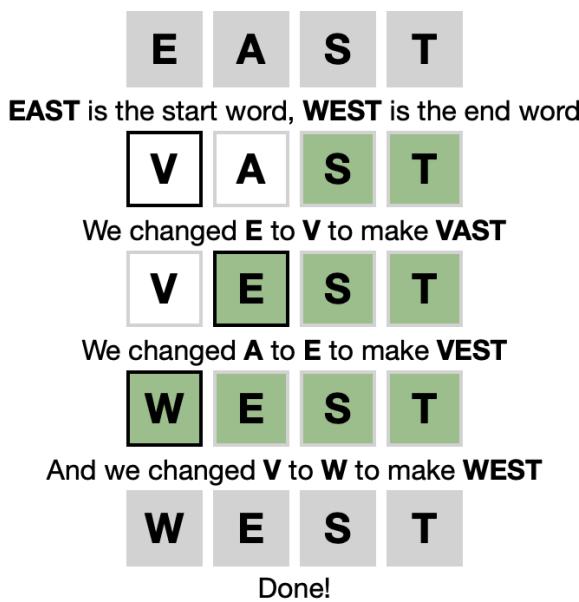
This game is called a "word ladder" and was invented by Lewis Carroll in 1877.

Rules

Weave your way from the start word to the end word.

Each word you enter **can only change 1 letter** from the word above it.

Example



Gambar 1.1 Ilustrasi dan Peraturan Permainan Word Ladder

Permainan ini tidak hanya menyediakan hiburan tetapi juga membantu dalam pengembangan keterampilan bahasa, seperti kemampuan untuk mengenali kata, memahami struktur kata, dan mengembangkan kosakata. Selain itu, Word Ladder juga sering digunakan dalam konteks pendidikan untuk mengajar tentang komponen bahasa dan pemecahan masalah.

BAB 2

TEORI SINGKAT

2.1. Algoritma Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah algoritma pencarian yang digunakan untuk menemukan jalur terpendek dari titik awal ke titik tujuan. UCS beroperasi dengan cara memperluas node dengan "biaya" terkecil terakumulasi dari titik awal, di mana biaya di sini bisa berupa jarak, waktu, atau metrik lain yang relevan. Algoritma ini tidak menggunakan heuristik dan menjamin menemukan solusi optimal. UCS sering digunakan dalam graf berbobot dan berarah, menjadikannya cocok untuk masalah yang memerlukan pemecahan berbasis biaya seperti pencarian rute terpendek.

2.2. Algoritma Greedy Best First Search

Greedy Best First Search adalah algoritma pencarian yang memprioritaskan ekspansi node yang paling menjanjikan untuk mencapai tujuan berdasarkan heuristik tertentu. Ini memilih node berdasarkan nilai heuristik yang mencerminkan estimasi terbaik menuju tujuan, tanpa mempertimbangkan biaya yang telah dikeluarkan. Algoritma ini biasanya lebih cepat daripada UCS karena kurangnya perhatian pada biaya yang telah dikeluarkan, tetapi tidak selalu menjamin solusi yang optimal. Algoritma ini sangat berguna untuk mendapatkan solusi cepat di mana keoptimalan bukanlah prioritas utama.

2.3. Algoritma A*

A* adalah algoritma pencarian yang menggabungkan kelebihan dari UCS dan Greedy Best First dengan menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$, di mana $g(n)$ adalah biaya dari node awal ke node n, dan $h(n)$ adalah estimasi biaya dari node n ke tujuan (heuristik). A* memperluas node dengan nilai $f(n)$ terendah, memastikan bahwa pencarian efisien dan cenderung ke solusi optimal asalkan heuristik $h(n)$ adalah admissible, yaitu tidak pernah melebih-lebihkan biaya sebenarnya untuk mencapai tujuan. A* adalah algoritma yang sangat populer dalam pathfinding dan graf berbobot.

BAB 3

ANALISIS & IMPLEMENTASI PROGRAM

3.1. Source Code Program

3.1.1. Algorithm.java

```
package satriadhikara.stima.tucil3.algorithm;

import satriadhikara.stima.tucil3.helper.Node;
import satriadhikara.stima.tucil3.model.InputWord;
import satriadhikara.stima.tucil3.model.Response;

import java.util.LinkedList;
import java.util.List;

public abstract class Algorithm {
    public abstract Response search(InputWord input);

    protected int getHeuristic(String current, String target) {
        int count = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) != target.charAt(i)) {
                count++;
            }
        }
        return count;
    }

    protected List<String> constructPath(Node node) {
        LinkedList<String> path = new LinkedList<>();
        Node current = node;
        while (current != null) {
            path.addFirst(current.word());
            current = current.parent();
        }
        return path;
    }
}
```

- **Class Algorithm**

Kelas ini berjenis kelas abstrak yang bakal di-*inherit* dengan kelas-kelas algoritma-algoritma.

- **Metode search**

Metode ini adalah metode abstrak yang harus diimplementasikan oleh kelas turunan. Metode ini digunakan untuk mencari berdasarkan input yang merupakan objek dari kelas InputWord.

- **Metode getHeuristic**

Metode ini digunakan untuk menghitung dan mengembalikan jumlah karakter yang berbeda antara dua string current dan target. Ini adalah fungsi heuristik yang

akan digunakan dalam algoritma pencarian berbasis heuristik seperti A* dan Greedy Best-First-Search.

- **Metode constructPath**

Metode ini digunakan untuk membangun jalur dari node yang diberikan ke root dari pohon (atau graf). Metode ini menggunakan LinkedList untuk menyimpan jalur, dan dimulai dari node yang diberikan, bergerak ke node induk sampai mencapai root (di mana induk adalah null). Kata-kata dari setiap node ditambahkan ke awal daftar, menghasilkan jalur dari root ke node yang diberikan

3.1.2. AStar.java

```
package satriadhikara.stima.tucil3.algorithm;

import satriadhikara.stima.tucil3.helper.Node;
import satriadhikara.stima.tucil3.model.InputWord;
import satriadhikara.stima.tucil3.model.Response;
import satriadhikara.stima.tucil3.helper.Dictionary;

import java.util.*;

public class AStar extends Algorithm {
    @Override
    public Response search(InputWord input) {
        String startWord = input.startWord();
        String targetWord = input.targetWord();

        PriorityQueue<Node> frontier = new
PriorityQueue<>(Comparator.comparingInt(Node::getTotalCost));
        Set<String> visited = new HashSet<>();

        frontier.add(new Node(startWord, null, 0,
getHeuristic(startWord, targetWord)));
        visited.add(startWord);

        int totalNodesVisited = 0;

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            totalNodesVisited++;

            if (current.word().equals(targetWord)) {
                return new Response("Path found",
constructPath(current), 0, totalNodesVisited, 0);
            }

            Set<String> neighbors =
Dictionary.getWordNeighbors(current.word(), visited);
            for (String neighbor : neighbors) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    int cost = current.cost() + 1;
                    int heuristic = getHeuristic(neighbor,
targetWord);
                    frontier.add(new Node(neighbor, current, cost,

```

```

        heuristic));
    }
}

return new Response("No path found", null, 0,
totalNodesVisited, 0);
}
}

```

- **Metode search**

Metode ini mengambil dua kata, kata awal dan kata tujuan, dari objek InputWord, memulai pencarian dengan memasukkan kata awal ke dalam antrian prioritas yang diurutkan berdasarkan total biaya (biaya akumulasi ditambah dengan heuristik). Fungsi ini memproses simpul dengan prioritas tertinggi, mengecek apakah itu mencapai kata tujuan. Jika ya, jalur yang ditemukan dan jumlah simpul yang dikunjungi dikembalikan. Jika tidak, fungsi mencari tetangga kata saat ini, mengupdate biaya dan heuristik mereka, dan memasukkan mereka ke antrian. Proses ini berulang hingga kata tujuan ditemukan atau antrian kosong, di mana fungsi akan mengembalikan pesan bahwa tidak ada jalur yang ditemukan, menunjukkan keefektifan A* dalam mencari solusi yang efisien dengan menggunakan heuristik untuk memandu pencarian.

3.1.3. GreedyBestFirstSearch.java

```

package satriadhikara.stima.tucil3.algorithm;

import satriadhikara.stima.tucil3.helper.Node;
import satriadhikara.stima.tucil3.model.InputWord;
import satriadhikara.stima.tucil3.model.Response;
import satriadhikara.stima.tucil3.helper.Dictionary;

import java.util.*;

public class GreedyBestFirstSearch extends Algorithm {
    @Override
    public Response search(InputWord input) {
        String startWord = input.startWord();
        String targetWord = input.targetWord();

        PriorityQueue<Node> frontier = new
PriorityQueue<>(Comparator.comparingInt(Node::heuristic));
        Set<String> visited = new HashSet<>();

        frontier.add(new Node(startWord, null, 0,
getHeuristic(startWord, targetWord)));
        visited.add(startWord);

        int totalNodesVisited = 0;

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            totalNodesVisited++;

```

```

        if (current.word().equals(targetWord)) {
            return new Response("Path found",
constructPath(current), 0, totalNodesVisited, 0);
        }

        Set<String> neighbors =
Dictionary.getWordNeighbors(current.word(), visited);
        for (String neighbor : neighbors) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                int heuristic = getHeuristic(neighbor,
targetWord);
                frontier.add(new Node(neighbor, current, 0,
heuristic));
            }
        }
    }

    return new Response("No path found", null, 0,
totalNodesVisited, 0);
}
}

```

- **Metode search**

Metode ini menerima input berupa objek InputWord yang menyediakan kata awal (startWord) dan kata tujuan (targetWord). Mulai dengan menambahkan kata awal ke dalam antrian prioritas (frontier) yang diurutkan berdasarkan heuristik dari setiap simpul. Set visited digunakan untuk melacak kata-kata yang telah dikunjungi. Selama antrian tidak kosong, simpul dengan heuristik terendah diambil, dan jika kata simpul tersebut cocok dengan kata tujuan, fungsi mengembalikan jalur yang ditemukan. Jika tidak, fungsi mencari tetangga dari kata tersebut yang belum dikunjungi, menghitung heuristik mereka, menambahkannya ke antrian, dan proses ini berulang sampai kata tujuan ditemukan atau antrian kosong, di mana fungsi akan mengembalikan bahwa tidak ada jalur yang ditemukan, mencatat total simpul yang dikunjungi.

3.1.4. UCS.java

```

package satriadhikara.stima.tucil3.algorithm;

import satriadhikara.stima.tucil3.helper.Node;
import satriadhikara.stima.tucil3.model.InputWord;
import satriadhikara.stima.tucil3.model.Response;
import satriadhikara.stima.tucil3.helper.Dictionary;

import java.util.*;

public class UCS extends Algorithm {

    @Override
    public Response search(InputWord input) {
        String start = input.startWord();

```

```

        String end = input.targetWord();

        Queue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(Node::cost));
        Set<String> visited = new HashSet<>();
        queue.add(new Node(start, null, 0, 0));
        visited.add(start);

        int totalNodesVisited = 0;

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            String currentWord = current.word();

            totalNodesVisited++;

            if (currentWord.equals(end)) {
                List<String> path = new ArrayList<>();
                int cost = -1;
                while (current != null) {
                    path.add(current.word());
                    cost = current.cost();
                    current = current.parent();
                }
                Collections.reverse(path);
                return new Response("Path found", path, cost,
totalNodesVisited, 0);
            }

            Set<String> neighbors =
Dictionary.getWordNeighbors(currentWord, visited);
            for (String neighbor : neighbors) {
                int newCost = current.cost() + 1;
                if (!visited.contains(neighbor)) {
                    queue.add(new Node(neighbor, current, newCost,
0));
                    visited.add(neighbor);
                }
            }
        }

        return new Response("No path found", null, 0,
totalNodesVisited, -1);
    }
}

```

- **Metode search**

Metode ini mengambil input berupa objek InputWord dengan kata awal (start) dan kata akhir (end). Fungsi ini memulai dengan memasukkan kata awal ke dalam antrian berprioritas (queue), yang diurutkan berdasarkan biaya dari setiap simpul. Set visited digunakan untuk melacak kata-kata yang sudah dikunjungi. Dalam prosesnya, fungsi mengambil simpul dengan biaya terendah, mengecek apakah kata tersebut sama dengan kata akhir. Jika sama, jalur dari kata awal ke kata akhir dibangun dan dikembalikan dengan total biaya. Jika tidak, fungsi akan mencari tetangga dari kata tersebut yang belum dikunjungi, menghitung biaya baru, menambahkan mereka ke antrian, dan menandai sebagai dikunjungi. Proses ini diulang hingga kata akhir

ditemukan atau antrian kosong, di mana fungsi akan mengembalikan bahwa tidak ada jalur yang ditemukan, mencatat total simpul yang dikunjungi dan biaya keseluruhan.

3.1.5. Node.java

```
package satriadhikara.stima.tucil3.helper;

import lombok.Getter;

public record Node(String word, @Getter
satriadhikara.stima.tucil3.helper.Node parent, int cost,
                    int heuristic) {
    public int getTotalCost() {
        return cost + heuristic;
    }
}
```

- Kelas Node ini adalah kelas yang dipakai dalam kelas algoritma-algoritma yang terdiri dari atribut word, parent, cost, dan heuristic. Untuk atribut heuristic hanya akan dipakai dalam algoritma A* dan Greedy Best First.

3.1.6. Dictionary.java

```
package satriadhikara.stima.tucil3.helper;

import java.io.*;
import java.util.*;

public class Dictionary {
    public static Set<String> words;

    static {
        try {
            words = loadWordsFromFile();
            System.out.println("Dictionary loaded successfully.");
        } catch (Exception e) {
            System.out.println("Error loading dictionary: " +
e.getMessage());
        }
    }

    private static Set<String> loadWordsFromFile() throws Exception {
        Set<String> words = new HashSet<>();
        try {
            File file = new
File("src/main/resources/words_oracle.txt");
            Scanner scanner = new Scanner(file);
            while (scanner.hasNextLine()) {
                String word = scanner.nextLine().trim();
                words.add(word);
            }
            scanner.close();
        } catch (FileNotFoundException e) {

```

```

        throw new Exception("File not found");
    }
    return words;
}

public static Set<String> getWordNeighbors(String currentWord,
Set<String> visited) {
    Set<String> neighbors = new HashSet<>();
    StringBuilder wordBuilder = new StringBuilder(currentWord);

    for (int i = 0; i < wordBuilder.length(); i++) {
        char originalChar = wordBuilder.charAt(i);
        for (char c = 'a'; c <= 'z'; c++) {
            if (c != originalChar) {
                wordBuilder.setCharAt(i, c);
                String newWord = wordBuilder.toString();
                if (words.contains(newWord) &&
!visited.contains(newWord)) {
                    neighbors.add(newWord);
                }
            }
        }
        wordBuilder.setCharAt(i, originalChar);
    }

    return neighbors;
}

public static boolean isWordExist(String word) {
    return words.contains(word);
}

public static boolean isOneLetterDifferent(String word1, String
word2) {
    if (word1.length() != word2.length()) {
        return false;
    }
    int count = 0;
    for (int i = 0; i < word1.length(); i++) {
        if (word1.charAt(i) != word2.charAt(i)) {
            if (++count > 1) {
                return false;
            }
        }
    }
    return count == 1;
}
}

```

- Kelas Dictionary ini adalah kelas yang bertanggung jawab atas menangani kata-kata dari *dictionary*. Kelas ini akan menge-load kata-kata dari file dictionary yang ada pas pertama kali dipanggil (dalam pertama kali program dimulai).
- Metode getWordNeighbors akan mengecek semua tetangga (yang beda satu huruf dalam kata tersebut) di kumpulan kata *dictionary*. Kalau ada bakal ditambah ke dalam set dan mengembalikan semua tetangganya.
- Metode isWordExist bakal mengembalikan benar kalau ada di dalam *dictionary*.
- Metode isOneLetterDifferent bakal mengembalikan benar kalau ada satu huruf yang beda diantara dua kata yang diinput.

- Kelas ini akan dipakai dalam semua program algoritma-algoritma untuk menentukan cost ataupun heuristic dan tetangganya.

3.1.7. Api.java

```

package satriadhikara.stima.tucil3.controller;

import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import satriadhikara.stima.tucil3.algorithm.AStar;
import satriadhikara.stima.tucil3.algorithm.GreedyBestFirstSearch;
import satriadhikara.stima.tucil3.algorithm.UCS;
import satriadhikara.stima.tucil3.helper.Dictionary;
import satriadhikara.stima.tucil3.model.InputWord;
import satriadhikara.stima.tucil3.model.Response;
import org.springframework.http.ResponseEntity;

@CrossOrigin
@RestController
public class Api {

    @GetMapping(value = "/api", produces = "application/json")
    public ResponseEntity<String> api(
        @RequestParam(value = "StartWord") String startWord,
        @RequestParam(value = "EndWord") String endWord,
        @RequestParam(value = "Method") String method) {
        startWord = startWord.toLowerCase();
        endWord = endWord.toLowerCase();
        if (!Dictionary.isWordExist(startWord)) {
            return new Response("StartWord is not a valid word",
null, -1, -1, -1).json();
        }
        if (!Dictionary.isWordExist(endWord)) {
            return new Response("EndWord is not a valid word",
null, -1, -1, -1).json();
        }
        if (!method.equals("AStar") && !method.equals("UCS") &&
!method.equals("GBFS")) {
            return new Response("Method is not valid", null, -1,
-1, -1).json();
        }
        if (startWord.length() != endWord.length()) {
            return new Response("Words length is not the same",
null, -1, -1, -1).json();
        }

        InputWord input = new InputWord(startWord, endWord);
        Response response;
        long startTime = System.currentTimeMillis();
        Runtime runtime = Runtime.getRuntime();
        long memoryBefore = runtime.totalMemory() -
runtime.freeMemory();
        if (method.equals("AStar")) {
            AStar aStar = new AStar();
            response = aStar.search(input);
        }
    }
}

```

```

        } else if (method.equals("UCS")) {
            UCS ucs = new UCS();
            response = ucs.search(input);
        } else {
            GreedyBestFirstSearch gbfs = new
            GreedyBestFirstSearch();
            response = gbfs.search(input);
        }
        long endTime = System.currentTimeMillis();
        long executionTime = endTime - startTime;
        long memoryAfter = runtime.totalMemory() -
        runtime.freeMemory();
        response.setExecutionTime(executionTime);
        long memoryUsed = Math.abs(memoryAfter - memoryBefore) /
        1024;
        response.setMemoryUsed(memoryUsed);
        return response.json();
    }
}

```

- Kelas Api ini adalah kelas yang meng-*handle endpoint* yang akan dipakai untuk berkomunikasi dengan *frontend*. *Endpoint* yang disediakan oleh kelas ini menerima 3 parameter *query* yaitu StartWord, EndWord, dan Method.
- Kelas ini akan melakukan validasi atas kata-kata yang dikasih oleh *client*, dan juga melakukan panggilan fungsi algoritma-algoritma yang sesuai dengan inputan *client*.
- Kelas ini akan merespon dengan JSON yang berisi jawaban (*path*) kalau ketemu dan error kalau tidak.

3.1.8. InputWord.java

```

package satriadhikara.stima.tucil3.model;

import jakarta.validation.constraints.NotBlank;

public record InputWord(@NotBlank String startWord, @NotBlank
String targetWord) {
    public InputWord(String startWord, String targetWord) {
        this.startWord = startWord;
        this.targetWord = targetWord;
    }
}

```

- Kelas InputWord ini adalah kelas yang bakal dipakai ketika kelas api dipakai dan untuk parameter metode search pada kelas-kelas algoritma.

3.1.9. Response.java

```

package satriadhikara.stima.tucil3.model;

import lombok.Setter;
import org.springframework.http.ResponseEntity;
import org.springframework.http.HttpStatus;

```

```

import java.util.List;

public class Response {
    private final String message;
    private final List<String> path;
    @Setter
    private long executionTime;
    private int nodeVisited;
    @Setter
    private long memoryUsed;

    public Response(String message, List<String> path, int executionTime, int nodeVisited, long memoryUsed) {
        this.message = message;
        this.path = path;
        this.executionTime = executionTime;
        this.nodeVisited = nodeVisited;
        this.memoryUsed = memoryUsed;
    }

    public ResponseEntity<String> json() {
        StringBuilder json = new StringBuilder("{}");
        json.append("\"message\":\"\"");
        if (path != null) {
            json.append(",\"path\":[\"");
            for (int i = 0; i < path.size(); i++) {

                json.append("\")").append(path.get(i).toUpperCase()).append("\")");
                if (i < path.size() - 1) {
                    json.append(",");
                }
            }
            json.append("],\"");
            json.append("\\"memoryUsed (KB)\":").append(memoryUsed);
        }
        json.append(",\"executionTime\":").append(executionTime);
        json.append(",\"nodeVisited\":").append(nodeVisited);
        json.append("}");

        if (path == null) {
            return new ResponseEntity<>(json.toString(), HttpStatus.BAD_REQUEST);
        } else {
            return new ResponseEntity<>(json.toString(), HttpStatus.OK);
        }
    }
}

```

- Kelas Response ini adalah kelas untuk dikasih kepada *request api*. Kelas response ini memiliki metode message yaitu untuk mengasih tahu apa yang terjadi, path yaitu jawabannya, executionTime yaitu berapa lama programnya berjalan dalam menyelesaikan solusi tersebut, nodeVisited adalah berapa banyak *node* yang dilalui oleh program dalam mencari solusi, memoryUsed adalah berapa banyak memori yang digunakan dalam program tersebut.
- Kelas ini memiliki konstruktor, dan metode json yaitu metode yang bakal digunakan dalam mengasih *response* kepada *client*.

3.1.10. BackendApplication.java

```
package satriadhikara.stima.tucil3;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import satriadhikara.stima.tucil3.helper.Dictionary;

@SpringBootApplication
public class BackendApplication {

    public static void main(String[] args) {
        SpringApplication.run(BackendApplication.class, args);
        Dictionary dictionary = new Dictionary();
    }
}
```

- Kelas ini adalah kelas *main* untuk menjalankan programnya (springboot) dan juga memanggil kelas Dictionary untuk mengeload kata-kata dari *dictionary* di awal.

3.2. Implementasi Bonus

Bonus GUI diimplementasikan dalam bentuk website yaitu menggunakan ReactJS-Vite-Typescript. GUI ini menerima 3 *input field*, yaitu input untuk startWord, endWord, dan metode algoritma yang mau dipakai. Setelah submit GUI akan menampilkan jawaban dari solusi yang sudah diimplementasi sesuai algoritmanya serta waktu dan berapa banyak *Node* yang dikunjungi, kalau tidak ada solusi GUI akan mengeluarkan error serta waktu eksekusinya dan juga berapa banyak *Node* yang dikunjungi.

3.3. Definisi dari $f(n)$ dan $g(n)$

3.1.1. UCS

$f(n)$: Total cost dari path dari node awal ke node n.

$g(n)$: Biaya aktual dari node start ke node n.

$$f(n) = g(n)$$

Dalam UCS, fungsi evaluasi $f(n)$ hanya bergantung pada $g(n)$, yang adalah total cost akumulatif dari start ke node n. UCS adalah algoritma yang berfokus pada biaya, menjelajahi node dengan total biaya terendah terlebih dahulu.

3.1.2. Greedy Best First Search

$f(n)$: Estimasi biaya dari node n ke tujuan (goal).

$h(n)$: Heuristic yang mengestimasi biaya terendah dari node n ke goal.

$$f(n) = h(n)$$

Dalam Greedy Best First Search, fokus utamanya adalah pada heuristik ($h(n)$), yang memberikan estimasi cost dari node n ke tujuan. Algoritma ini tidak mempertimbangkan cost

yang telah dikeluarkan ($g(n)$), sehingga cenderung lebih cepat namun kurang optimal karena hanya berfokus pada goal tanpa mempertimbangkan total cost perjalanan.

3.1.3. A*

$f(n)$: Estimasi total biaya terkecil dari node start ke goal melalui node n.

$g(n)$: Biaya aktual dari node start ke node n.

$h(n)$: Heuristic yang mengestimasi biaya dari node n ke goal.

$$f(n) = g(n) + h(n)$$

A* menggabungkan aspek UCS dan Greedy Best First Search. Fungsi $f(n)$ pada A* adalah jumlah dari $g(n)$, biaya aktual dari start ke n, dan $h(n)$, estimasi biaya dari n ke goal. Algoritma A* mencoba menemukan balance antara path cost yang telah dikeluarkan dan estimasi cost ke tujuan.

3.4. Apakah heuristik yang digunakan pada algoritma A* admissible?

Heuristik pada algoritma A* dianggap admissible jika selalu memperkirakan biaya yang lebih rendah dari biaya sebenarnya dari node n ke tujuan ($h(n)$ lebih kecil dari $h^*(n)$). Jika heuristik adalah admissible, maka algoritma A* dijamin untuk menemukan solusi optimal. Heuristik juga harus konsisten, yang berarti biaya yang diperkirakan dari satu node ke tujuan tidak melebihi total biaya langkah dari node tersebut ke node lain ditambah dengan perkiraan biaya dari node kedua ke tujuan.

3.5. Pada kasus word ladder, apakah algoritma UCS sama dengan BFS?

Dalam konteks Word Ladder, algoritma Uniform Cost Search (UCS) dan Breadth-First Search (BFS) bisa menghasilkan urutan node yang dibangkitkan dan path yang sama jika semua langkah memiliki biaya yang sama, misalnya 1 per langkah. UCS, yang biasanya mempertimbangkan bobot atau biaya untuk memilih node berikutnya, akan berperilaku seperti BFS ketika semua biaya sisi adalah sama, karena di situ prioritas hanya diberikan pada urutan pencarian tanpa mempertimbangkan biaya. Namun, pada umumnya UCS dirancang untuk menangani bobot yang berbeda-beda, sedangkan BFS selalu mengasumsikan bobot yang seragam atau tidak mempertimbangkan bobot sama sekali.

3.6. Secara teoritis, apakah algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada kasus word ladder?

A* menggunakan kombinasi dari biaya jalur yang telah ditempuh ($g(n)$) dan estimasi biaya ke tujuan ($h(n)$) untuk membentuk fungsi $f(n) = g(n) + h(n)$. Dengan heuristik yang tepat, A* mampu lebih cepat mendekati solusi optimal karena fokus pada rute yang secara heuristik lebih menjanjikan daripada hanya mengexplore semua rute secara merata seperti dalam UCS.

UCS beroperasi dengan mengexpand node berdasarkan biaya dari start node, mengabaikan informasi estimasi ke tujuan. Dalam word ladder, ini bisa berarti UCS menghabiskan waktu yang signifikan menjelajahi pilihan yang kurang relevan terhadap tujuan.

Dalam kasus word ladder, A* bisa lebih efisien dari UCS karena A* memanfaatkan heuristik untuk mengarahkan pencarian. Jika heuristik tersebut baik (admissible dan ideal), pencarian bisa lebih cepat karena A* fokus pada node yang paling berpotensi mengarah ke solusi dengan meminimalkan biaya total. Heuristik yang baik mengurangi jumlah node yang perlu dijelajahi.

3.7. Secara teoritis, apakah algoritma *Greedy Best First Search* menjamin solusi optimal untuk persoalan *word ladder*?

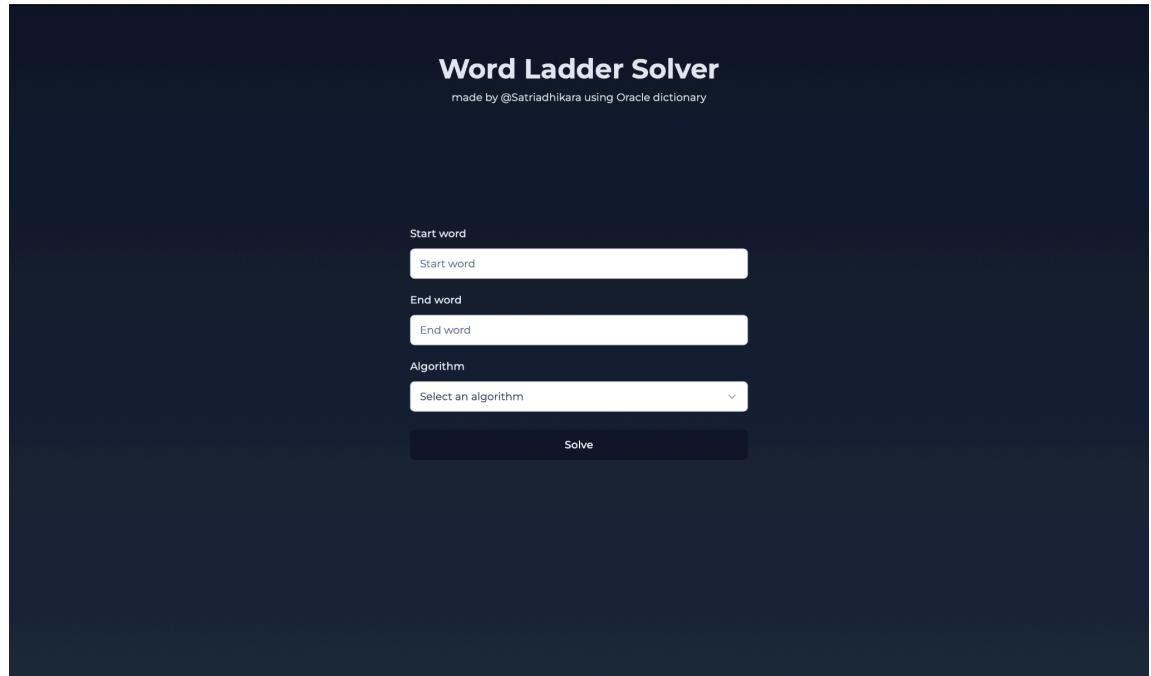
Greedy Best First Search bekerja dengan selalu memilih node yang paling menjanjikan berdasarkan heuristik yang diberikan, mengabaikan biaya total yang telah ditempuh ($g(n)$). Karena heuristiknya hanya memfokuskan pada estimasi ke tujuan ($h(n)$), Greedy bisa sangat cepat tetapi sering kali mengabaikan jalur yang lebih pendek atau lebih murah secara keseluruhan karena terpaku pada pemilihan yang tampak optimal per langkah.

Ini bisa menyebabkan Greedy Best First Search mengabaikan jalur yang awalnya tampak kurang menjanjikan tetapi memiliki biaya total yang lebih rendah, sehingga algoritma bisa terjebak pada solusi suboptimal. Dalam konteks word ladder, pendekatan ini bisa mengabaikan jalur alternatif yang pada awalnya tidak tampak menarik tetapi akhirnya mengarah ke solusi yang lebih efisien.

BAB 4

EKSPERIMEN

4.1. Tampilan GUI

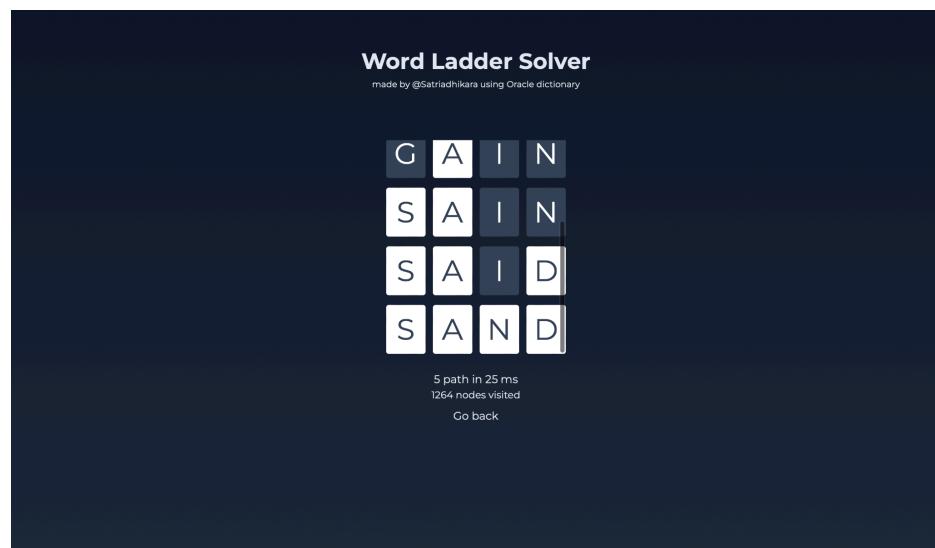


Gambar 4.1 Tampilan GUI Website

4.2. Uniform Cost Search

1. StartWord = GRIT

EndWord = SAND



Gambar 4.2.1 Hasil test case UCS 1

2. StartWord = HELLO

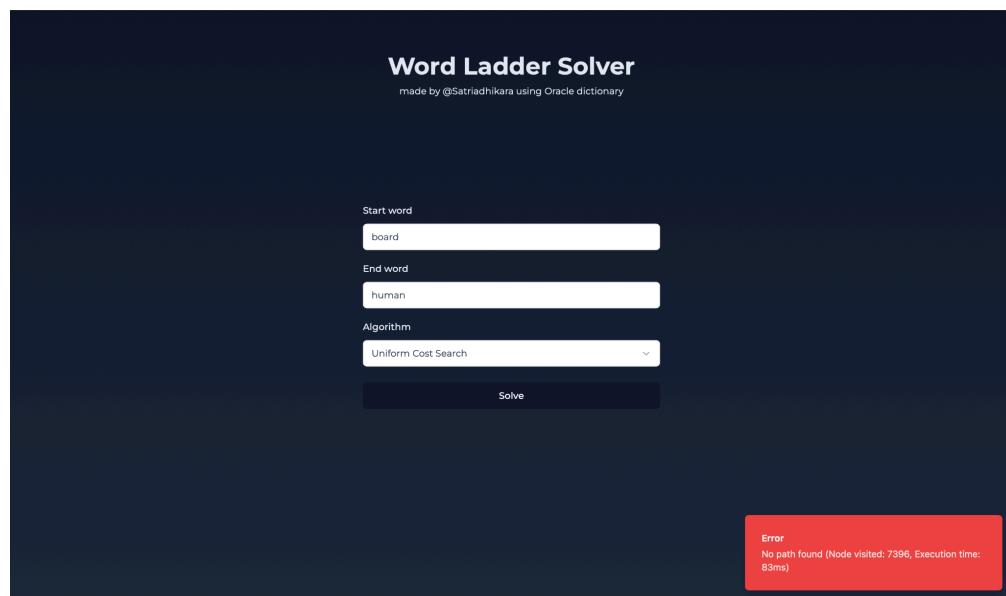
EndWord = WATCH



Gambar 4.2.2 Hasil test case UCS 2

3. StartWord = BOARD

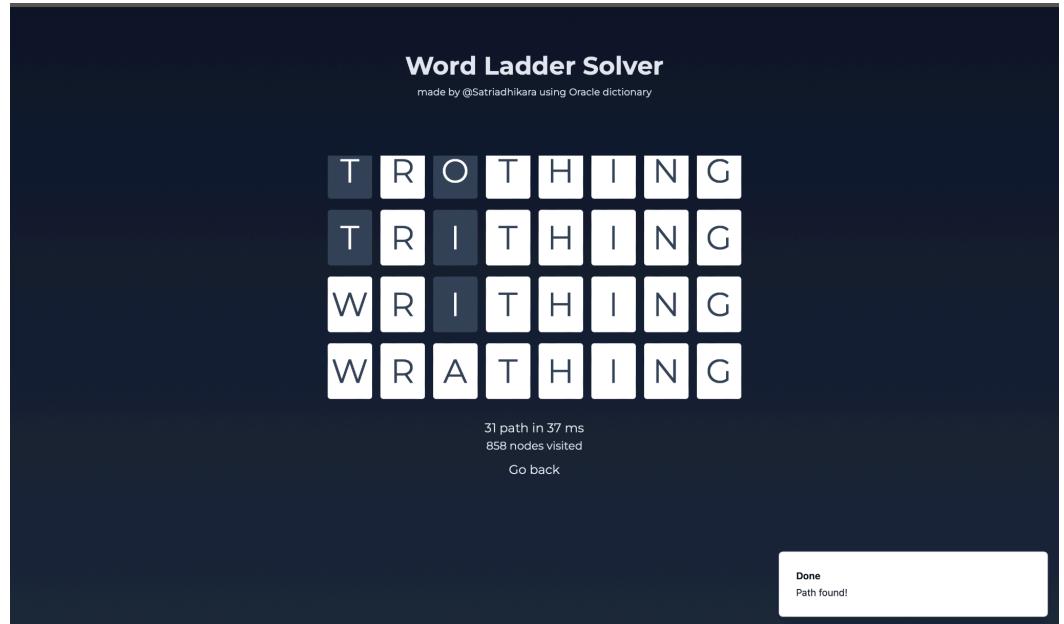
EndWord = HUMAN



Gambar 4.2.3 Hasil test case UCS 3

4. StartWord = QUIRKING

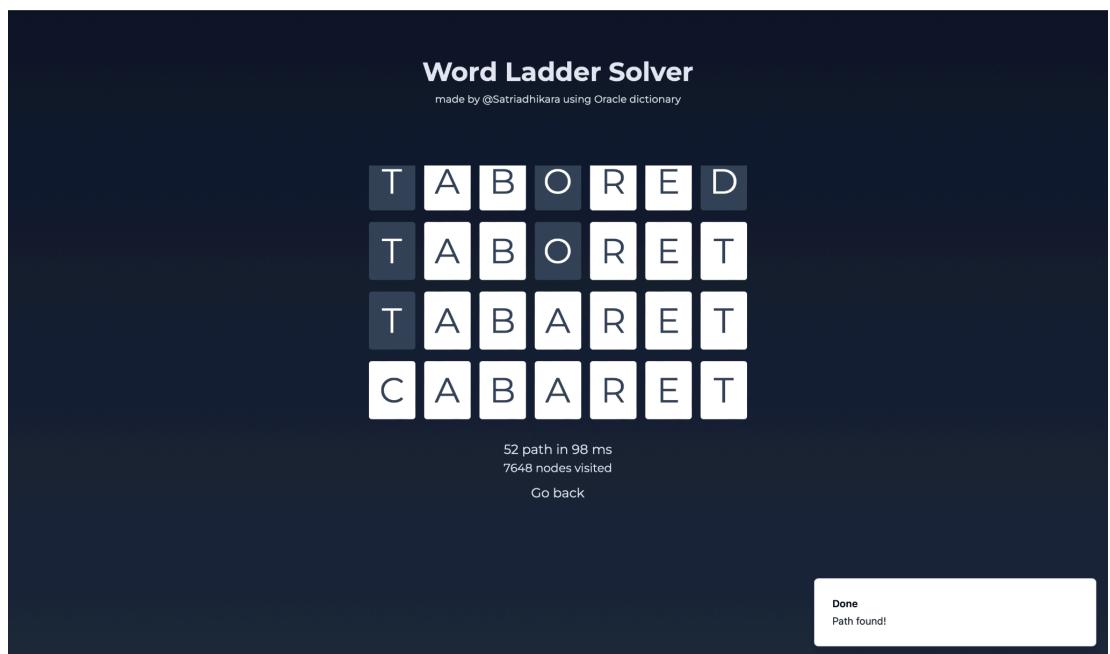
EndWord = WRATHING



Gambar 4.2.4 Hasil test case UCS 4

5. StartWord = ATLASES

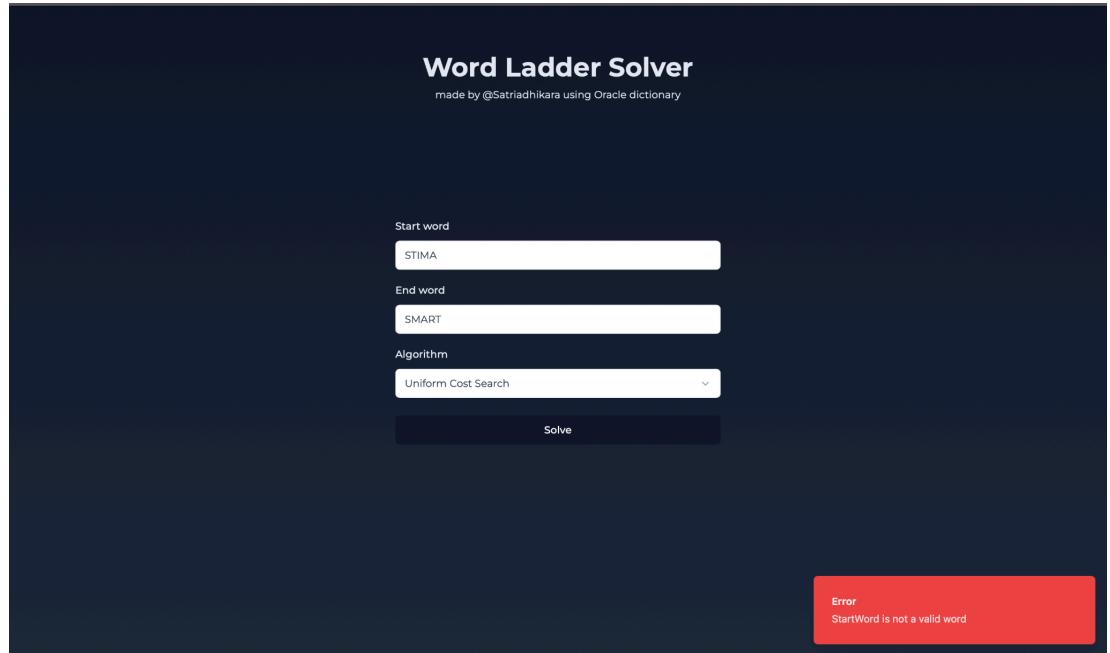
EndWord = CABARET



Gambar 4.2.5 Hasil test case UCS 5

6. StartWord = STIMA

EndWord = SMART

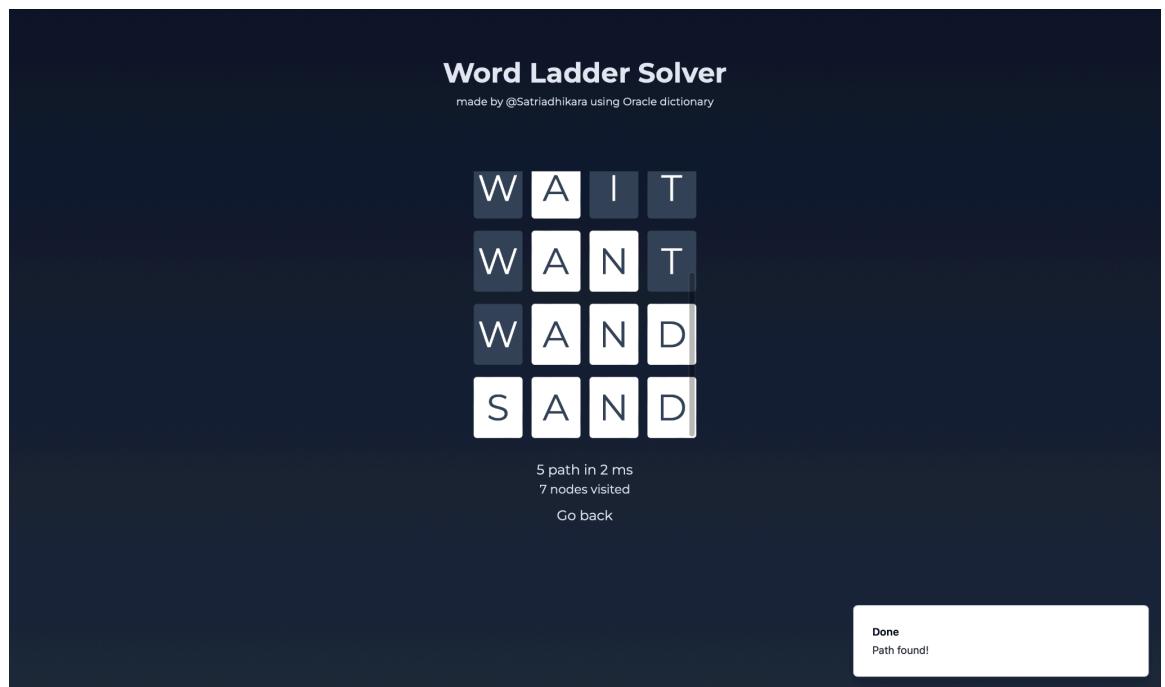


Gambar 4.2.6 Hasil test case UCS 6

4.3. Greedy Best First Search

1. StartWord = GRIT

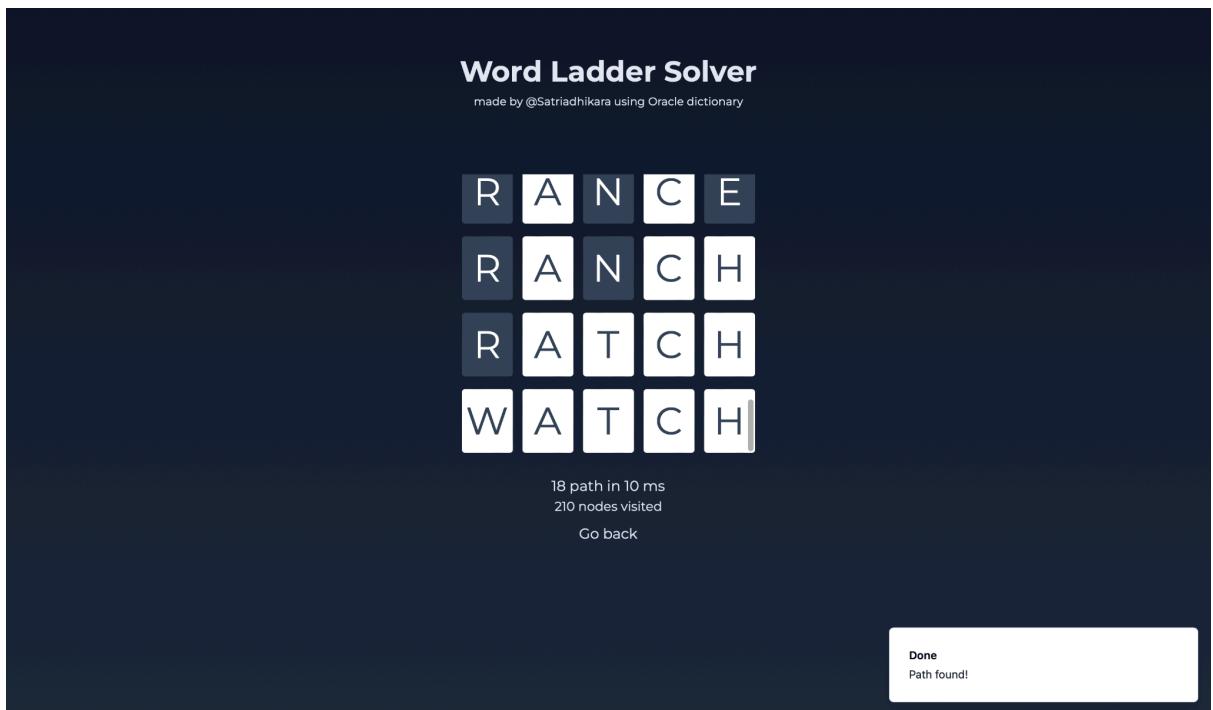
EndWord = SAND



Gambar 4.3.1 Hasil test case GBFS 1

2. StartWord = HELLO

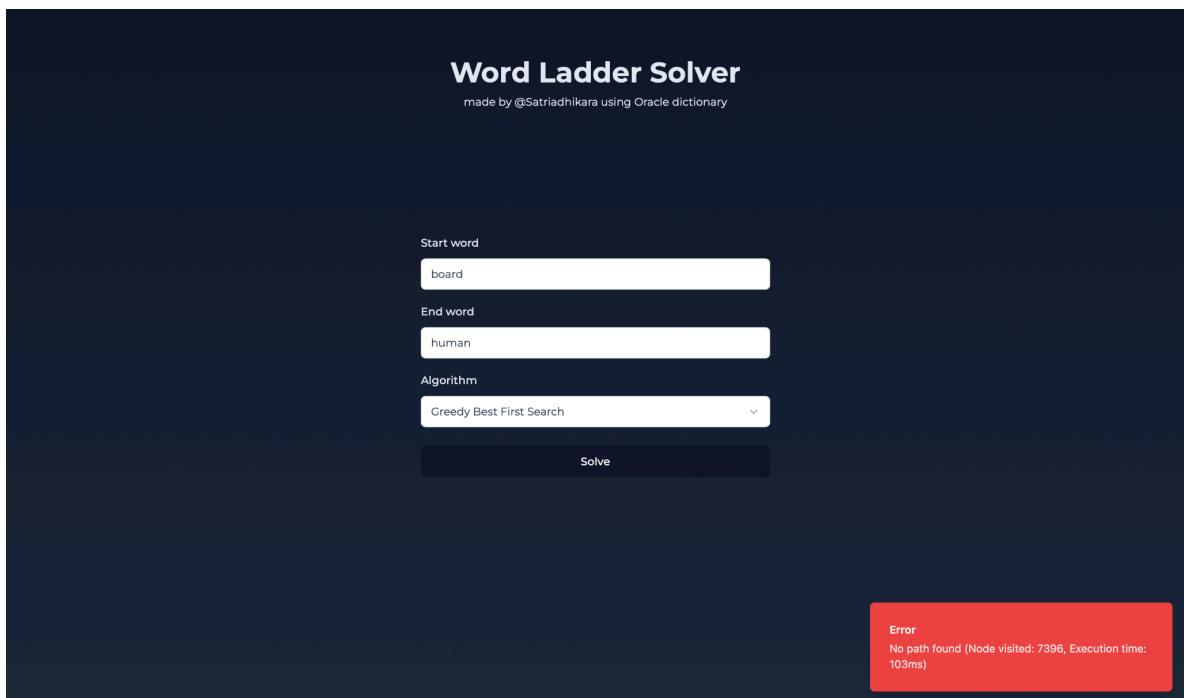
EndWord = WATCH



Gambar 4.3.2 Hasil test case GBFS 2

3. StartWord = BOARD

EndWord = HUMAN



Gambar 4.3.3 Hasil test case GBFS 3

4. StartWord = QUIRKING

EndWord = WRATHING



Gambar 4.3.4 Hasil test case GBFS 4

5. StartWord = ATLASES

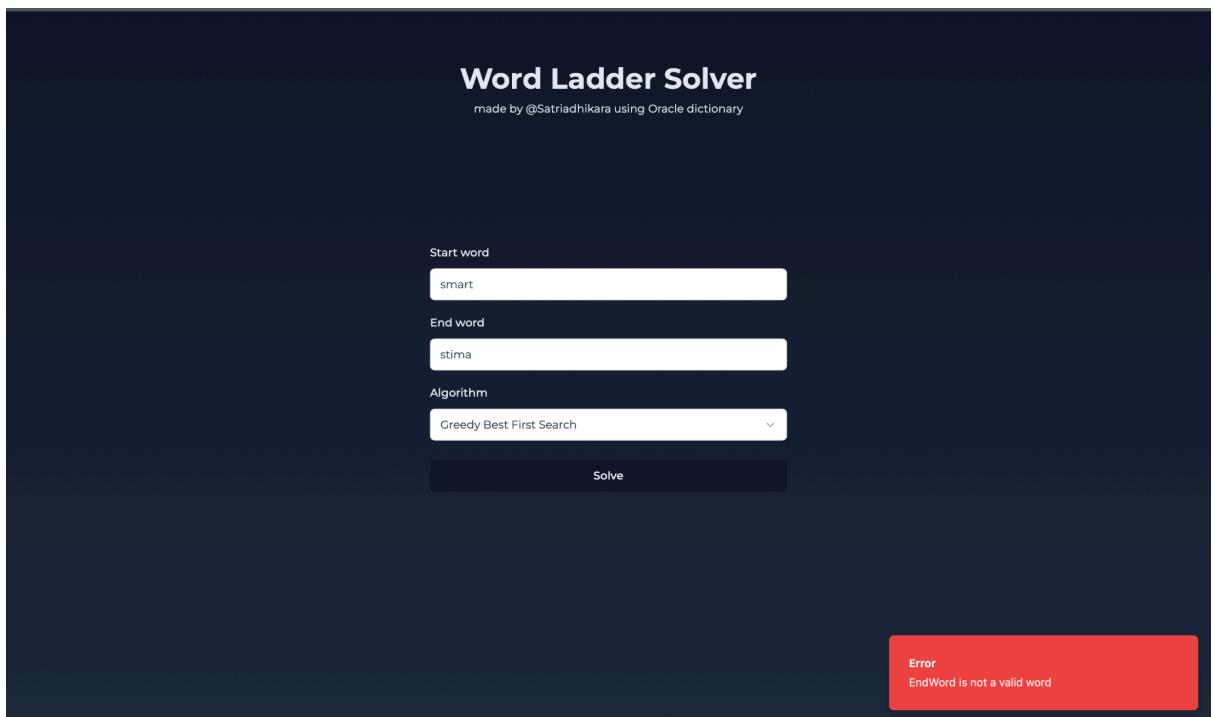
EndWord = CABARET



Gambar 4.3.5 Hasil test case GBFS 5

6. StartWord = SMART

EndWord = STIMA

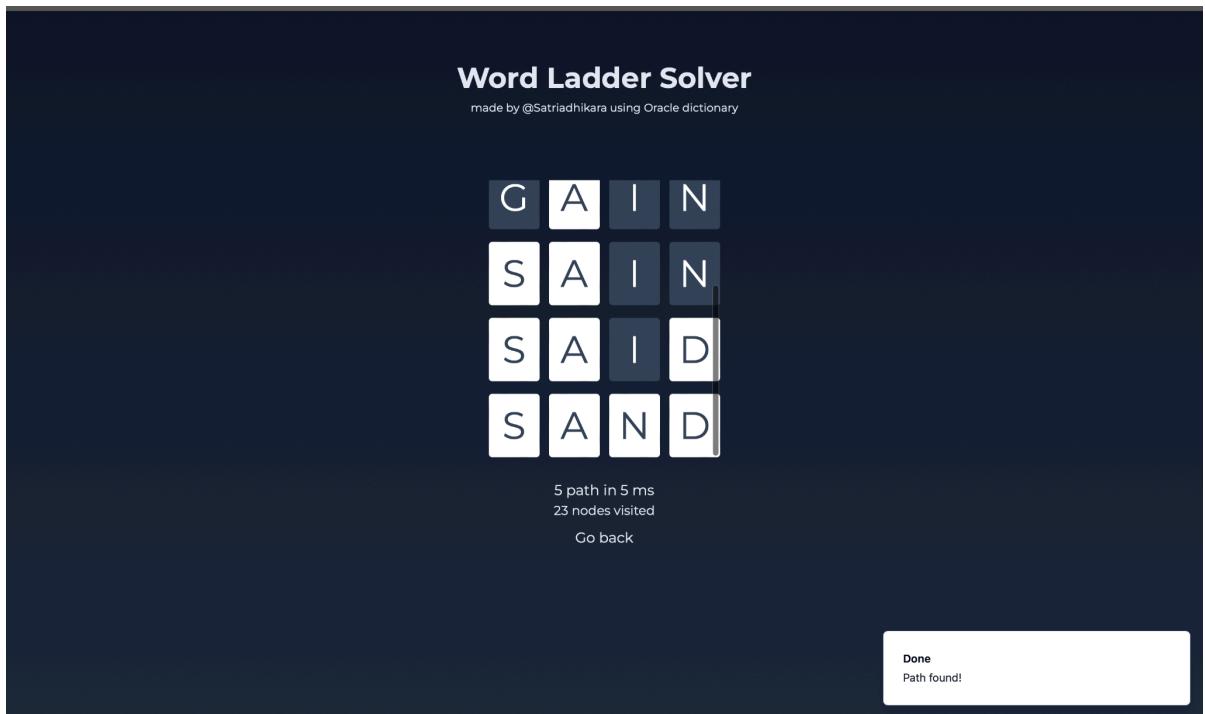


Gambar 4.3.6 Hasil test case GBFS 6

4.4. A*

1. StartWord = GRIT

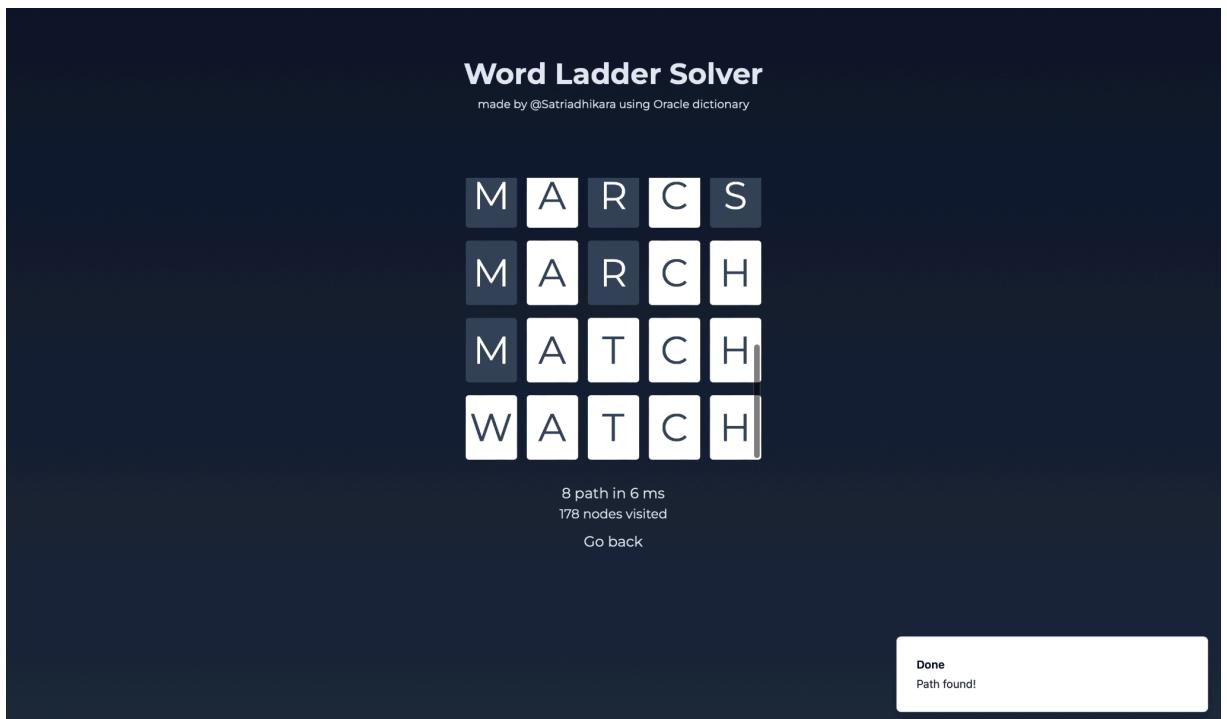
EndWord = SAND



Gambar 4.4.1 Hasil test case A* 1

2. StartWord = HELLO

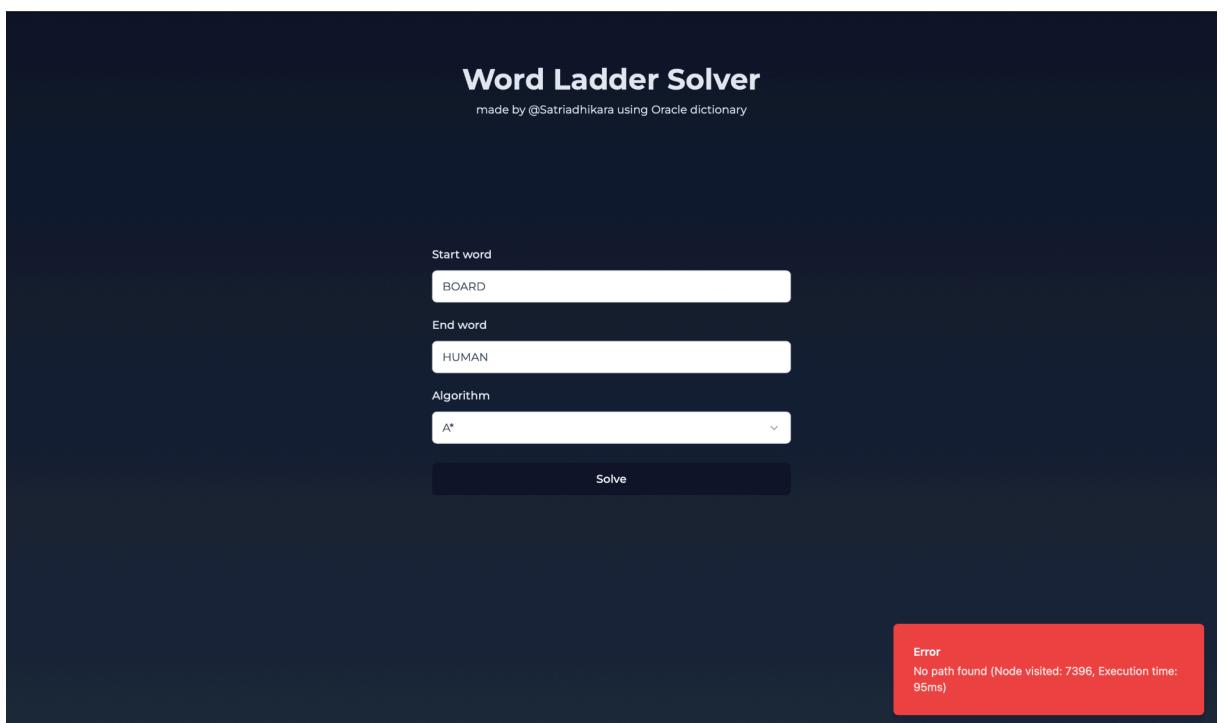
EndWord = WATCH



Gambar 4.4.2 Hasil test case A* 2

3. StartWord = BOARD

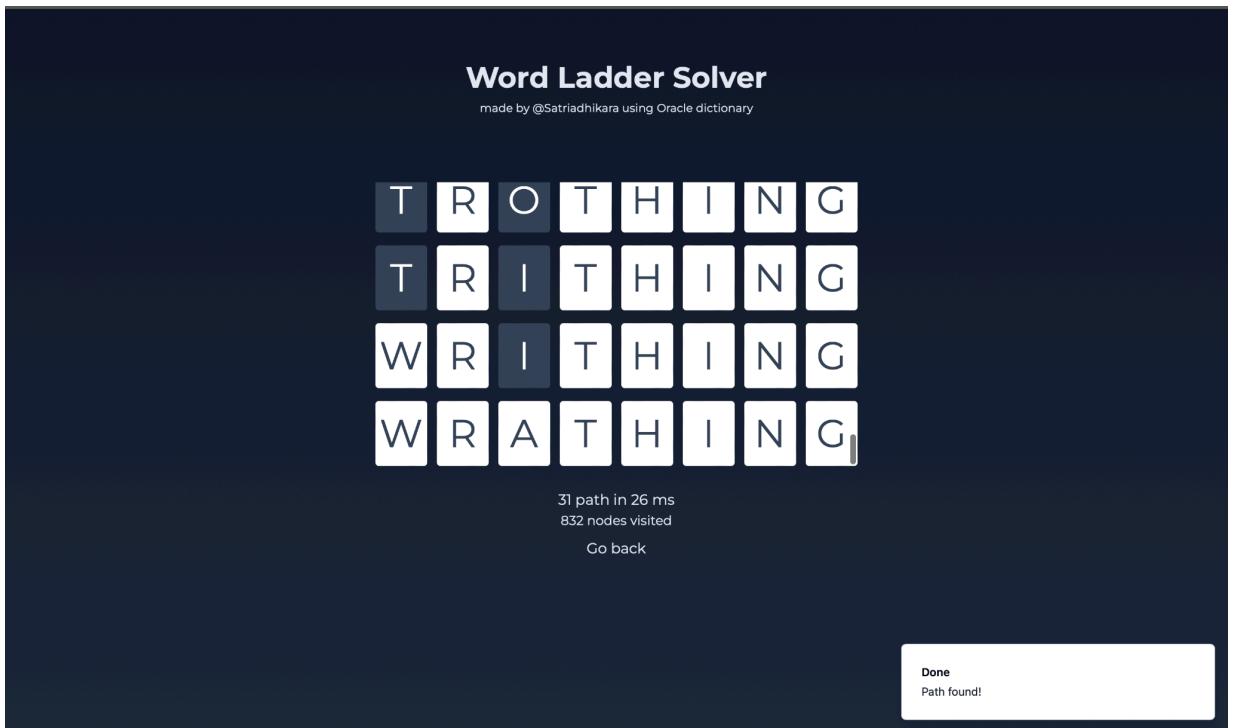
EndWord = HUMAN



Gambar 4.4.3 Hasil test case A* 3

4. StartWord = QUIRKING

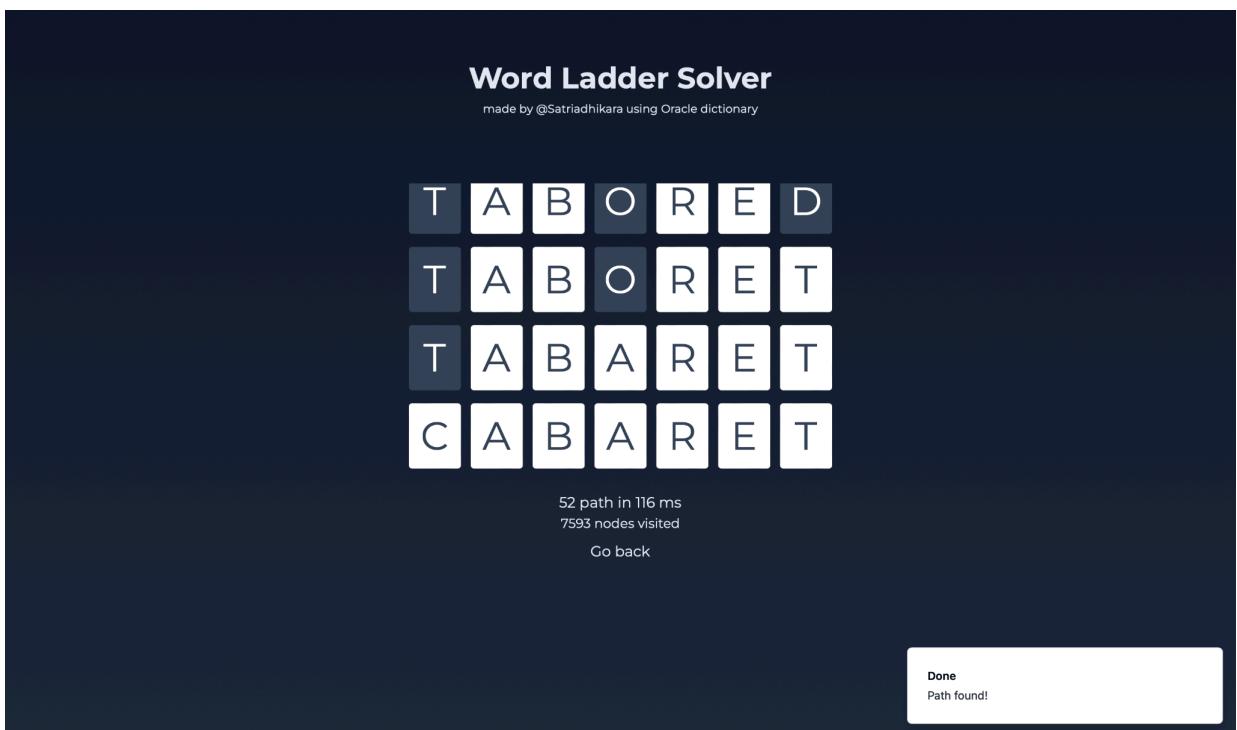
EndWord = WRATHING



Gambar 4.4.4 Hasil test case A* 4

5. StartWord = ATLASES

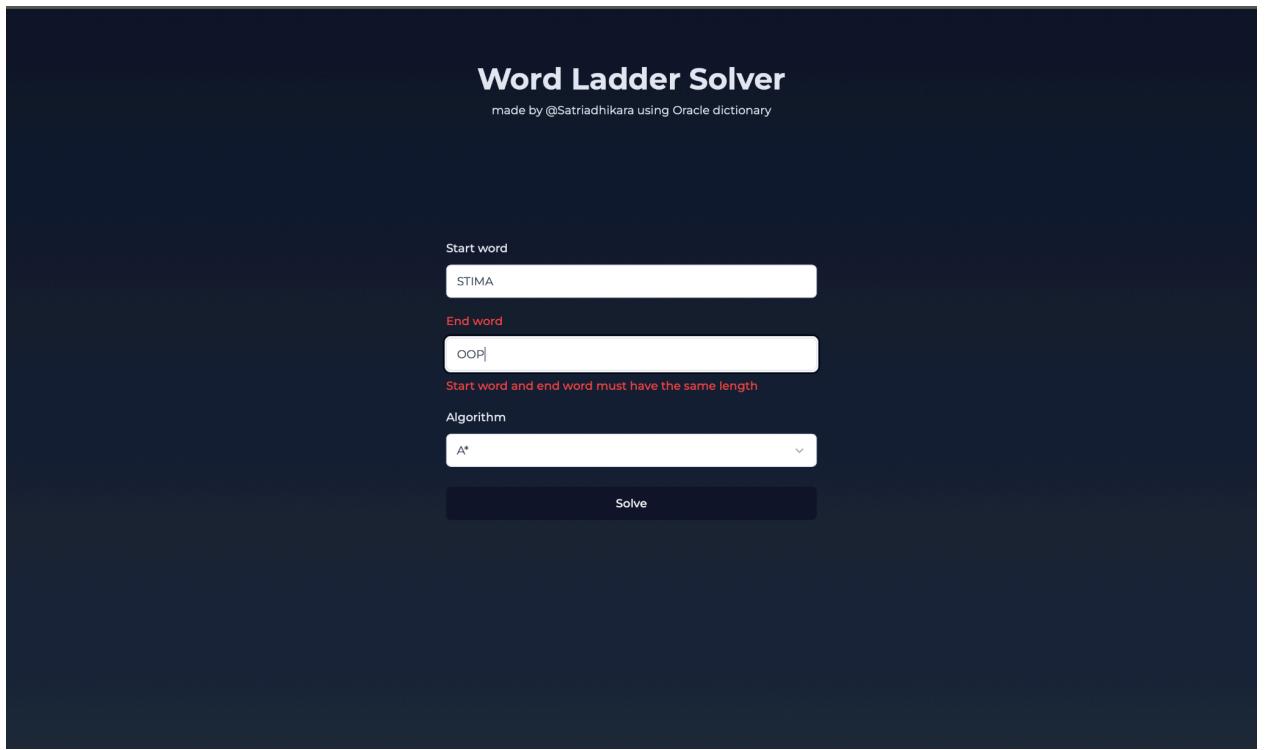
EndWord = CABARET



Gambar 4.4.5 Hasil test case A* 5

6. StartWord = STIMA

EndWord = OOP



Gambar 4.4.6 Hasil test case A* 6

BAB 5

PENUTUP

1. Kesimpulan

Dalam mencari solusi untuk game *Word Ladder*, algoritma A* merupakan pilihan terbaik jika heuristik yang tepat dan admissible dapat didefinisikan. Ini memberikan keseimbangan antara kecepatan dan keakuratan, dengan meminimalkan risiko eksplorasi jalur yang tidak efisien sambil menjaga fokus pada jalur yang potensial mengarah pada solusi optimal. Greedy Best First Search bisa digunakan untuk solusi cepat tetapi kurang optimal, sedangkan UCS dan BFS cocok untuk skenario dengan biaya seragam dan struktur data sederhana.

2. Saran

Dalam mengembangkan solusi algoritma untuk game Word Ladder, disarankan untuk memilih antara kecepatan dan ketepatan sesuai kebutuhan: A* untuk ketepatan optimal dengan heuristik yang admissible, dan Greedy Best First Search untuk kecepatan dengan kelemahan pada keoptimalan.

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/satriadhikara/Tucil3_13522125

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	✓	