Laporan Tugas Kecil 3 IF2211 Strategi Algoritma

Disusun oleh:

Dewantoro Triatmojo (13522011)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA INSTITUT TEKNOLOGI BANDUNG

2024

Daftar Isi

BAB 1: Deskripsi Singkat

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.

Informed Search sering disebut juga dengan Heuristic Search. Pencarian dengan algoritma ini menggunakan knowledge yang spesifik kepada permasalahan yang dihadapi disamping dari definisi masalahnya itu sendiri. Metode ini mampu menemukan solusi secara lebih efisien daripada yang bisa dilakukan pada metode uninformed strategy. Pada solver permainan word ladder ini digunakan tiga algoritma informed search, yaitu Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A* search.

BAB 2: Algoritma Program

Pada dasarnya, implementasi algoritma search untuk ketiga algoritma sama, namun yang membedakan hanyalah fungsi perhitungan cost `f(n)` untuk node suatu node.

Secara garis besar, algoritma search program mengikuti

1. Pertama, menginisialisasi priority queue untuk menyimpan urutan simpul ekspan (simpul yang akan di dequeue dan diperluas). Queue ini berisi node yang mana menyimpan data kata, cost, depth, dan parent node.

2. Program menginisialisasi semua kata di database kamus yang akan digunakan dalam algoritma search (usable dictionary) yaitu kata dalam kamus dengan panjang yang sama dengan input start word atau end word karena panjang kata tebakan pasti sama.

3. Program menginisialisasi sebuah HashMap dengan key String (kata) dan value boolean untuk menyimpan data apakah suatu kata sudah pernah dikunjungi atau belum sebelumnya.

4. Program menginisialisasi total node traversed = 0 (menyimpan jumlah node yang sudah dikunjungi) dan current best solution = null (menyimpan node terakhir dari current best found solution)

5. Program akan menghitung cost f(n) untuk node pertama (root) dan menginisialisasikan node pertama dengan isi word = start word, cost yang sebelumnya dihitung, depth = 0 dan parent node = null. Setelah itu, node root akan dimasukkan ke dalam priority queue.

6. Selagi priority queue tidak kosong,

a. Dequeue elemen node dari priority queue. Node ini merupakan node dengan cost terkecil pada queue.

b. Jika node yang baru di dequeue sebelumnya sudah pernah dikunjungi, lanjutkan (continue) loop. Hal ini bisa dicek menggunakan HashMap yang sudah diinisialisasi pada step nomor 3.

c. Jika current best solution sudah ketemu dan cost dari node yang baru didequeue lebih dari cost dari node ujung (leaf) dari current best solution, maka lanjutkan (continue) loop.

d. Perbarui data HashMap bahwa node yang baru didequeue sudah dikunjungi dan jumlah node yang dikunjungi ditambah satu.

e. Jika node yang baru didequeue memiliki kata yang sama dengan end word (solusi), maka bila solusi belum ditemukan sebelumnya atau jika sebelumnya sudah ditemukan namun cost dari node yang baru didequeue lebih kecil daripada cost ujung node dari current best solution, maka perbarui current best solution.

f. Jika node yang baru didequeue tidak memiliki kata yang sama dengan end word (belum solusi), maka ekspan simpul tersebut. Untuk setiap kata di usable dictionary yang belum visited dan hanya memiliki beda 1 huruf (sesuai aturan permainan) dengan kata pada simpul yang sedang di ekspan, hitung cost baru dan depth baru untuk node tersebut dan masukkan ke priority queue.

g. Lakukan a sampai priority queue kosong

Perhitungan fungsi cost f(n) mengikuti

1. Uniform Cost Search (UCS):

Pada uniform cost search, `f(n)` (cost untuk suatu node) mengikuti `f(n) = g(n)`. Dimana `g(n)` adalah jumlah langkah yang telah dilakukan diukur dari root node, atau dalam kata lain depth node dari root node.

2. Greedy Best First Search (GBFS)

   Pada greedy best first search, `f(n)` (cost untuk suatu node) mengikuti `f(n) = h(n)`. Dimana heuristic `h(n)` adalah estimasi cost untuk mencapai target word. Estimasi yang bisa kita gunakan adalah jumlah perbedaan karakter antara target word dengan current word, karena setiap step hanya boleh mengubah 1 karakter.

3. A* Search

   Pada A* search, `f(n)` (cost suatu node) mengikuti `f(n) = g(n) + h(n)`. Sama seperti sebelumnya, `g(n)` adalah jumlah langkah yang telah dilakukan diukur dari root node, atau dalam kata lain depth node dari root node dan h(n) adalah heuristic menggunakan jumlah perbedaan karakter antara target word dengan current word. Heuristic yang digunakan admissible karena tidak pernah melebihi jumlah langkah dari word ke target word `h*(n)`. Hal ini bisa dilihat karena jumlah perbedaan karakter merupakan jumlah langkah paling minimum untuk berpindah dari suatu word ke target word dengan hanya 1 step (bisa jadi lebih dari ini, melalui kata perantara).

Pada dasarnya, algoritma UCS sama saja seperti algoritma BFS dalam urutan pembangkitan nodenya. Hal ini dikarenakan pada UCS, f(n) = g(n) dimana g(n) adalah depth node. Artinya, saat melakukan ekspansi simpul, simpul-simpul baru akan memiliki cost currentDepth + 1 (depth yang sama memiliki cost yang sama). Artinya, urutan pemilihan di queuenya akan sama saja seperti BFS.

Secara teoritis, algoritma A* lebih efisien dibandingkan UCS. Hal ini dikarenakan A* juga menggunakan heuristic untuk mengukur jarak current node ke target word untuk memilih elemen selanjutnya untuk dikunjungi. Tidak seperti UCS, yang hanya

menggunakan depth dari tree. Dengan melakukan hal ini, A* cenderung memperluas simpul yang lebih mungkin mengarah ke tujuan, sehingga berpotensi menghindari penjelajahan jalur yang tidak diperlukan.

Secara teoritis, algoritma Greedy Best First Search (GBFS) tidak menjamin solusi optimal karena penggunaan algoritma greedy. Algoritma GBFS bahkan bisa saja stuck di local minima sehingga menyebabkan solusi tidak ditemukan.

BAB 3: Source Code

Penjelasan mengenai class, method, dan attribute tertera pada komentar program di gambar.

3.1 Class Main

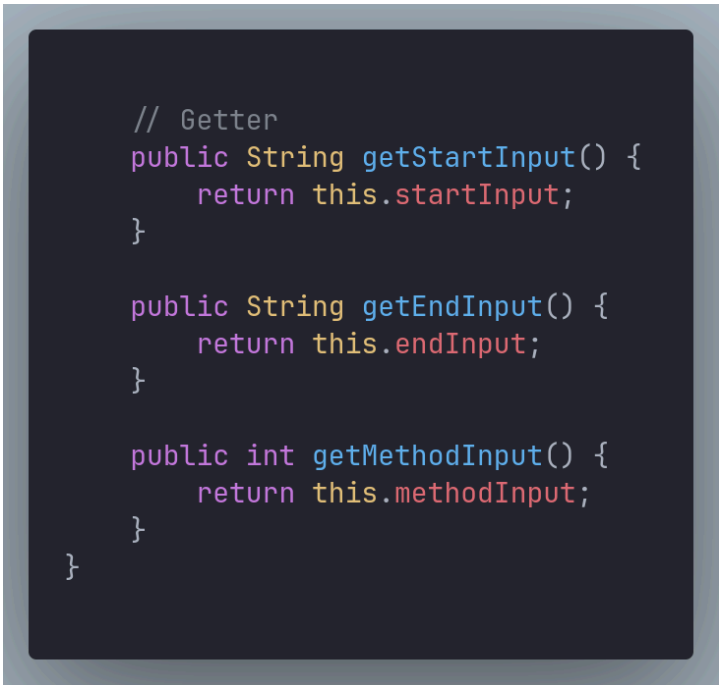
```java
import dictionary.Dictionary;
import input.Input;
import output.Output;
import solve.Solve;

public class Main {
    public static void main(String[] args) {
        // Print welcome message
        Output.printWelcome();

        // Read dictionary from file
        Dictionary.initializeDictionary();

        // Get input values
        // input object stores the input values
        Input input = new Input();
        input.initializeInputValue();

        // Get the solution
        // solve object stores the solution values
        Solve solve = new Solve();
        solve.calculateSolution(input);

        // Output result
        Output.printResult(solve);
    }
}
```

Gambar 3.1: Class Main (Program utama)

3.2 Class Input

```java
public class Input {
    // Attributes
    private String startInput;
    private String endInput;
    private int methodInput;

    // Intiialize the input
    public void initializeInputValue() {
        System.out.println(
                "=============================================================================================================================");

        // Initialize scanner
        Scanner sc = new Scanner(System.in);

        // Get start word
        System.out.println();
        System.out.print("Enter the start word: ");
        this.startInput = sc.nextLine().toLowerCase();
        // Validate start word
        while (!Utils.isAlphabetic(this.startInput) // Check if the start word is alphabetic
                || this.startInput.length() == 0 // Check if the start word is not empty
                || !Dictionary.isWordInDictionary(startInput) // Check if the start word is in the dictionary
        ) {
            // Message for invalid input
            if (!Utils.isAlphabetic(this.startInput))
                System.out.println("The start word must be alphabetic!");
            else if (this.startInput.length() == 0)
                System.out.println("The start word must not be empty!");
            else if (!Dictionary.isWordInDictionary(startInput))
                System.out.println("The start word must not be in the dictionary!");

            // Get new start word
            System.out.println();
            System.out.print("Enter the start word: ");
            this.startInput = sc.nextLine().toLowerCase();
        }

        // Get end word
        System.out.println();
        System.out.print("Enter the end word: ");
        this.endInput = sc.nextLine().toLowerCase();

        // Validation end word
        while (!Utils.isAlphabetic(this.endInput) // Check if the end word is alphabetic
                || this.endInput.length() != this.startInput.length() // Check end word length vs the start word
                || !Dictionary.isWordInDictionary(endInput) // Check if the end word is in the dictionary
                || this.startInput.equals(this.endInput) // Check if the end word is not the same as the start word
        ) {
            // Message
            if (!Utils.isAlphabetic(this.endInput))
                System.out.println("The end word must be alphabetic!");
            else if (this.endInput.length() != this.startInput.length())
                System.out.println("The end word must have the same length as the start word!");
            else if (!Dictionary.isWordInDictionary(endInput))
                System.out.println("The end word must be in the dictionary!");
            else if (this.startInput.equals(this.endInput))
                System.out.println("The end word must not be the same as the start word!");

            // Get new end word
            System.out.println();
            System.out.print("Enter the end word: ");
            this.endInput = sc.nextLine().toLowerCase();
        }

        // Get method
        // 1. Uniform Cost Search (UCS)
        // 2. Greedy Best First Search (GBFS)
        // 3. A* Search
        System.out.println();
        System.out.println("Choose the method: ");
        System.out.println("1. Uniform Cost Search (UCS)");
        System.out.println("2. Greedy Best First Search (GBFS)");
        System.out.println("3. A* Search");
        System.out.print("Enter the method: ");
        String methodInputStr = sc.nextLine();

        // Validate method
        // Must be 1, 2, or 3 and a integer
        while (!Utils.isNumeric(methodInputStr) // Check if the method is a number
                || (Integer.parseInt(methodInputStr) < 1 || Integer.parseInt(methodInputStr) > 3) // 1 ⩽ x ⩽ 3
        ) {
            // Message
            if (!Utils.isNumeric(methodInputStr))
                System.out.println("The method must be a number!");
            else if (Integer.parseInt(methodInputStr) < 1 || Integer.parseInt(methodInputStr) > 3)
                System.out.println("The method must be 1, 2, or 3!");

            // Get new method
            System.out.println();
            System.out.print("Enter the method: ");
            methodInputStr = sc.nextLine();
        }
        this.methodInput = Integer.parseInt(methodInputStr);

        // Close scanner
        sc.close();

        System.out.println();
        System.out.println(
                "=============================================================================================================================");
    }
```

Gambar 3.2: Definisi atribut dan method initializeInputValue() pada class Input



```
    // Getter
    public String getStartInput() {
        return this.startInput;
    }

    public String getEndInput() {
        return this.endInput;
    }

    public int getMethodInput() {
        return this.methodInput;
    }
}
```

Gambar 3.3: Getter untuk mendapatkan atribut startInput, endInput, and methodInput.

3.3 Class Dictionary

```java
package dictionary;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Scanner;

public class Dictionary {
    // Dictionary map (to validate words with O(1) complexity)
    public static Map<String, Boolean> dictionaryMap;

    // Constructor
    public static void initializeDictionary() {
        // Initialize dictionary
        dictionaryMap = new HashMap<>();

        // Load dictionary from file
        try {
            File file = new File("./src/dictionary/words_alpha.txt");
            Scanner sc = new Scanner(file);
            while (sc.hasNextLine()) {
                String word = sc.nextLine();
                dictionaryMap.put(word, true);
            }
            sc.close();
        } catch (FileNotFoundException e) {
            // File not found
            System.out.println("An error occurred: file not found");
            e.printStackTrace();
            System.exit(0);
        }
    }

    // Check if word is in dictionary
    public static boolean isWordInDictionary(String word) {
        return dictionaryMap.containsKey(word);
    }

    // Generate usable words from dictionary
    public static List<String> generateUsableDictionaryWords(int length) {
        List<String> usableDictionaryWords = new ArrayList<>();

        // Iterate through dictionary map
        dictionaryMap.keySet().forEach(word -> {
            if (word.length() == length) {
                usableDictionaryWords.add(word);
            }
        });

        return usableDictionaryWords;
    }
}
```

Gambar 3.4: Class Dictionary

3.4 Class Node

```java
package solve.node;

import java.util.Comparator;

public class Node {
    private String word; // The word that the node represents
    private int cost; // The cost of the node
    // The depth of the node in the search tree (o(1) access rather than o(d)
    // traversal to get depth )
    private int depth;
    private Node parent; // The parent of the node

    // Constructor
    public Node(String word, int cost, int depth, Node parent) {
        this.word = word;
        this.cost = cost;
        this.depth = depth;
        this.parent = parent;
    }

    // Getters
    public String getWord() {
        return word;
    }

    public int getCost() {
        return cost;
    }

    public int getDepth() {
        return depth;
    }

    public Node getParent() {
        return parent;
    }

    // Print the node (to help debug)
    public void printNode() {
        System.out.println("Word: " + word + " Cost: " + cost);
    }

    // Comparator to compare nodes based on their cost
    public static Comparator<Node> customComparator = new Comparator<Node>() {
        @Override
        public int compare(Node n1, Node n2) {
            return n1.getCost() - n2.getCost();
        }
    };
}
```

Gambar 3.5: Struktur data node

3.5 Class Search

```java
package solve.search;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;

import dictionary.Dictionary;
import lib.Utils;
import solve.node.Node;

public abstract class Search {
    // Store solution node and total node traversed
    private Node solutionNode;
    private int totalNodeTraversed;

    // Abstract method to be implemented by subclasses
    public abstract int calculateCost(int currentDepth, String word, String endWord);

    // Search algorithm
    // Algorithm is same, only calculateCost() is different.
    public void search(String startWord, String endWord) {
        // Initialize empty priority queue
        PriorityQueue<Node> pq = new PriorityQueue<>(Node.customComparator);

        // Get all usable words from the dictionary
        List<String> usableDictionaryWords = Dictionary.generateUsableDictionaryWords(startWord.length());

        // Initialize map to store visited nodes
        Map<String, Boolean> visited = new HashMap<>();
        for (String word : usableDictionaryWords) {
            visited.put(word, false);
        }

        // Initialize start node
        int startDepth = 0;
        int startCost = calculateCost(startDepth, startWord, endWord);
        Node startNode = new Node(startWord, startCost, startDepth, null);

        // Initialize solution data
        this.totalNodeTraversed = 0;
        this.solutionNode = null;

        // Add start node to priority queue
        pq.add(startNode);

        // Loop until priority queue is empty
        while (!pq.isEmpty()) {
            // Get node with the lowest cost
            Node lowestCostNodeDequeue = pq.poll();

            // Check if node is already visited before
            if (visited.get(lowestCostNodeDequeue.getWord())) {
                continue;
            }

            // Check if current best solution is found, the cost is more than the current
            // solution then continue
            if (this.solutionNode != null) {
                if (lowestCostNodeDequeue.getCost() > this.solutionNode.getCost()) {
                    continue;
                }
            }

            // Mark node as visited
            visited.put(lowestCostNodeDequeue.getWord(), true);
            this.totalNodeTraversed++;

            // Check if node is the end word
            if (lowestCostNodeDequeue.getWord().equals(endWord)) {
                // Check if current best solution is found
                if (this.solutionNode == null || lowestCostNodeDequeue.getCost() < this.solutionNode.getCost()) {
                    this.solutionNode = lowestCostNodeDequeue;
                }
            } else {
                // Update queue
                for (String word : usableDictionaryWords) {
                    // Only add words that is not yet visited and has only 1 different character
                    if (!visited.get(word)
                            && Utils.countDifferentCharacters(lowestCostNodeDequeue.getWord(), word) == 1) {
                        // Update queue with new nodes
                        int newDepth = lowestCostNodeDequeue.getDepth() + 1;
                        int newCost = calculateCost(newDepth, word, endWord);
                        Node newNode = new Node(word, newCost, newDepth, lowestCostNodeDequeue);
                        pq.add(newNode);
                    }
                }
            }
        }
    }
```

Gambar 3.6: Abstract class Search dan implementasi method `search()`

```
    // Getters
    // Get total node traversed
    public int getTotalNodeTraversed() {
        return this.totalNodeTraversed;
    }

    // Get solution
    public List<String> getSolution() {
        List<String> solution = new ArrayList<>();

        Node currentNode = this.solutionNode;
        while (currentNode != null) {
            solution.add(0, currentNode.getWord()); // Reverse the order (because we are traversing from end to start)
            currentNode = currentNode.getParent();
        }

        return solution;
    }
}
```

Gambar 3.7: Getters class Search

```
package solve.search;

public class UCS extends Search {
    // Constructor
    public UCS() {
        super();
    }

    // Calculate cost f(n) = g(n)
    // g(n) = depth
    @Override
    public int calculateCost(int currentDepth, String word, String endWord) {
        return currentDepth;
    }
}
```

Gambar 3.8: Class UCS dan implementasi fungsi `calculateCost()`

```java
package solve.search;

import lib.Utils;

public class GBFS extends Search {
    // Constructor
    public GBFS() {
        super();
    }

    // Calculate cost f(n) = h(n)
    // h(n) = heuristic
    @Override
    public int calculateCost(int depth, String word, String endWord) {
        int heuristic = Utils.countDifferentCharacters(word, endWord);

        return heuristic;
    }
}
```

Gambar 3.9: Class GBFS dan implementasi fungsi `calculateCost()`

```java
package solve.search;

import lib.Utils;

public class AStar extends Search {
    // Constructor
    public AStar() {
        super();
    }

    // Calculate cost f(n) = g(n) + h(n)
    // g(n) = depth
    // h(n) = heuristic
    @Override
    public int calculateCost(int depth, String word, String endWord) {
        int heuristic = Utils.countDifferentCharacters(word, endWord);

        return depth + heuristic;
    }
}
```

Gambar 3.10: Class AStar dan implementasi fungsi `calculateCost()`

3.6 Class Solve

```java
package solve;

import java.util.ArrayList;
import java.util.List;

import input.Input;
import solve.search.AStar;
import solve.search.GBFS;
import solve.search.UCS;

public class Solve {
    // Store solution
    private List<String> solution;
    private int totalNodeTraversed;
    private double duration;
    private long memoryUsed;

    // Constructor
    public Solve() {
        this.solution = new ArrayList<>();
        this.totalNodeTraversed = 0;
        this.duration = 0;
    }

    // Calculate solution
    public void calculateSolution(Input inputValue) {
        // Get start word
        String startWord = inputValue.getStartInput();

        // Get end word
        String endWord = inputValue.getEndInput();

        // Get method
        int method = inputValue.getMethodInput();

        // Get start time
        long startTime = System.nanoTime();

        // Solve
        if (method == 1) {
            // Using UCS
            UCS ucs = new UCS();
            ucs.search(startWord, endWord);
            this.solution = ucs.getSolution();
            this.totalNodeTraversed = ucs.getTotalNodeTraversed();
        } else if (method == 2) {
            // Solve using GBFS
            GBFS gbfs = new GBFS();
            gbfs.search(startWord, endWord);
            this.solution = gbfs.getSolution();
            this.totalNodeTraversed = gbfs.getTotalNodeTraversed();
        } else if (method == 3) {
            // Solve using A*
            AStar aStar = new AStar();
            aStar.search(startWord, endWord);
            this.solution = aStar.getSolution();
            this.totalNodeTraversed = aStar.getTotalNodeTraversed();
        }

        // Get end time
        long endTime = System.nanoTime();

        // Calculate duration
        this.duration = (endTime - startTime) / 1000000.0;

        // Calculate memory usage
        Runtime rt = Runtime.getRuntime();
        long memory = rt.totalMemory() - rt.freeMemory();
        System.out.println("Memory used: " + memory / 1024 + " KB");
        this.memoryUsed = memory / 1024;
    }

    // Getters
    // Get solution
    public List<String> getSolution() {
        return this.solution;
    }

    // Get total node traversed
    public int getTotalNodeTraversed() {
        return this.totalNodeTraversed;
    }

    // Get duration
    public double getDuration() {
        return this.duration;
    }

    // Get memory used
    public long getMemoryUsed() {
        return this.memoryUsed;
    }
}
```

Gambar 3.11: Class Solve

3.7 Class Output

```java
package output;

import java.util.List;

import solve.Solve;

public abstract class Output {
    // Method to print result values
    public static void printResult(final Solve result) {
        System.out.println(
                "========================================================================================================================");
        System.out.println();

        // Words
        List<String> solution = result.getSolution();
        if (solution.size() == 0) {
            System.out.println("Result: No solution found!");
        } else {
            System.out.println("Result: ");
            for (int i = 0; i < result.getSolution().size(); i++) {
                System.out.println(i + 1 + ". " + result.getSolution().get(i));
            }
        }

        // Total node traversed
        System.out.println("Total node traversed: " + result.getTotalNodeTraversed());

        // Duration
        System.out.println("Duration: " + result.getDuration() + " ms");

        // Memory used
        System.out.println("Memory used: " + result.getMemoryUsed() + " KB");

        System.out.println();
        System.out.println(
                "========================================================================================================================");
        System.out.println(
                "========================================================================================================================");
    }
```

Gambar 3.12: Class Output dan implementasi method `printResult()`

```java
    // Method to print welcome message
    public static void printWelcome() {
        System.out.println(
                "========================================================================================================================");
        System.out.println(
                "========================================================================================================================");
        System.out.println(
                "____    __    ____  ______   .______       _______      __          ___       _______   _______   _______ .______");
        System.out.println(
                "\\   \\  /  \\  /   / /  __  \\  |   _  \\     |       \\    |  |        /   \\     |       \\ |       \\ |   ____||   _  \\ ");
        System.out.println(
                " \\   \\/    \\/   / |  |  |  | |  |_)  |    |  .--.  |   |  |       /  ^  \\    |  .--.  ||  .--.  ||  |__   |  |_)  |");
        System.out.println(
                "  \\            /  |  |  |  | |      /     |  |  |  |   |  |      /  /_\\  \\   |  |  |  ||  |  |  ||   __|  |      / ");
        System.out.println(
                "   \\    /\\    /   |  `--'  | |  |\\  \\----.|  '--'  |   |  `----./  _____  \\  |  '--'  ||  '--'  ||  |____ |  |\\  \\----.");
        System.out.println(
                "    \\__/  \\__/     \\______/  | _| `._____||_______/    |_______/__/     \\__\\ |_______/ |_______/ |_______|| _| `._____|");
        System.out.println(
                "                                                                                                                        ");
        System.out.println(
                "                                 _______.  ______    __         ____    ____  _______ .______                          ");
        System.out.println(
                "                                /       | /  __  \\  |  |       \\   \\  /   / |   ____||   _  \\                          ");
        System.out.println(
                "                               |   (----`|  |  |  | |  |        \\   \\/   /  |  |__   |  |_)  |                         ");
        System.out.println(
                "                                \\   \\    |  |  |  | |  |         \\      /   |   __|  |      /                          ");
        System.out.println(
                "                            .----)   |   |  `--'  | |  `----.     \\    /    |  |____ |  |\\  \\----.                     ");
        System.out.println(
                "                            |_______/     \\______/  |_______|      \\__/     |_______|| _| `._____|                     ");
        System.out.println();
        System.out.println(
                "========================================================================================================================");
    }
}
```

Gambar 3.13: Class Output dan implementasi method `printWelcome()`

3.8 Class Utils

```java
package lib;

public abstract class Utils {
    // Check if the string is numeric
    public static boolean isNumeric(final String str) {
        try {
            Integer.parseInt(str);
            return true;
        } catch (NumberFormatException e) {
            return false;
        }
    }

    // Check if string is alphabetic
    public static boolean isAlphabetic(final String str) {
        return str.matches("[a-zA-Z]+");
    }

    // Check how many characters are different between two strings
    // Initial state: Both strings have the same length
    public static int countDifferentCharacters(final String str1, final String str2) {
        int count = 0;
        for (int i = 0; i < str1.length(); i++) {
            if (str1.charAt(i) != str2.charAt(i)) {
                count++;
            }
        }
        return count;
    }
}
```

Gambar 3.14: Class Utils

Bab 4: Uji Coba

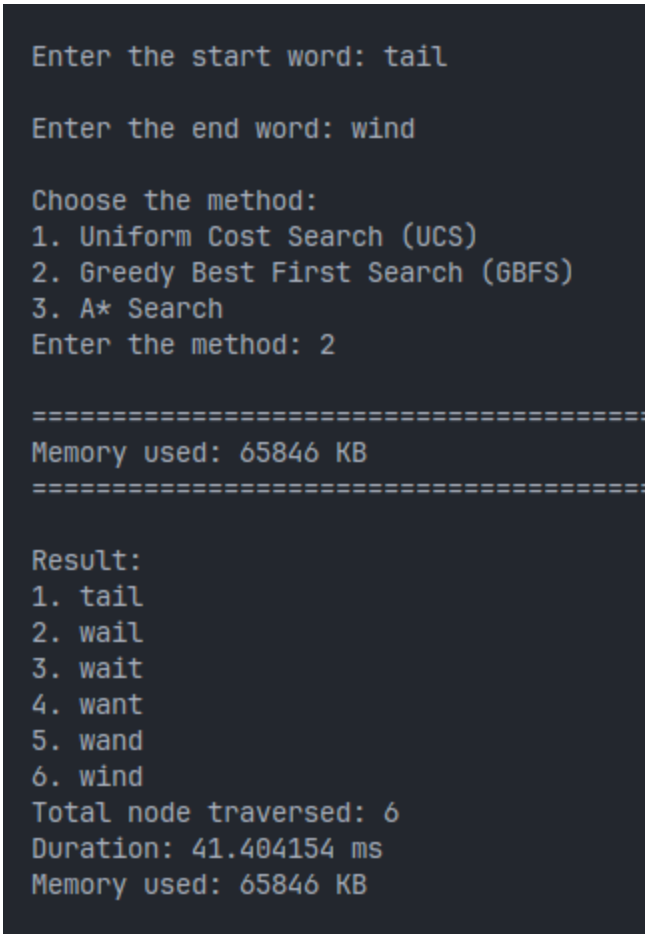
4.1 Test Case 1

a. Algoritma UCS

```
Enter the start word: tail

Enter the end word: wind

Choose the method:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Enter the method: 1

========================================
Memory used: 66629 KB
========================================

Result:
1. tail
2. tall
3. wall
4. will
5. wild
6. wind
Total node traversed: 5982
Duration: 908.113255 ms
Memory used: 66629 KB
```

Gambar 4.1: Test case 1 Algoritma UCS

b. Algoritma GBFS



Gambar 4.2: Test case 1 Algoritma GBFS

c. Algoritma A* Search

```
Enter the start word: tail

Enter the end word: wind

Choose the method:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Enter the method: 3

=======================================
Memory used: 62906 KB
=======================================

Result:
1. tail
2. wail
3. wait
4. want
5. wand
6. wind
Total node traversed: 66
Duration: 88.673967 ms
Memory used: 62906 KB
```

Gambar 4.3: Test case 1 Algoritma A* Search

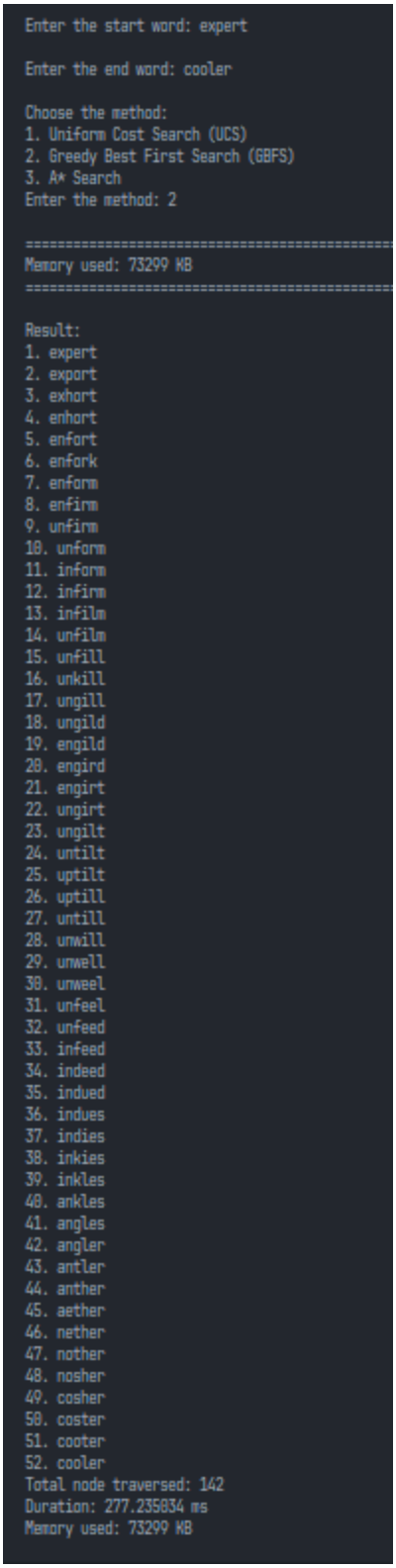
4.2 Test Case 2

a. Algoritma UCS

```
Enter the start word: expert

Enter the end word: cooler

Choose the method:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Enter the method: 1

========================================
Memory used: 64560 KB
========================================

Result:
1. expert
2. export
3. exhort
4. enhort
5. enfort
6. enfork
7. unfork
8. uncork
9. uncore
10. encore
11. endore
12. endere
13. enders
14. eiders
15. ciders
16. coders
17. cooers
18. cooees
19. cooeed
20. cooled
21. cooler
Total node traversed: 4648
Duration: 4897.254172 ms
Memory used: 64560 KB
```

Gambar 4.4: Test case 2 Algoritma UCS

b. Algoritma GBFS



Gambar 4.5: Test case 2 Algoritma GBFS

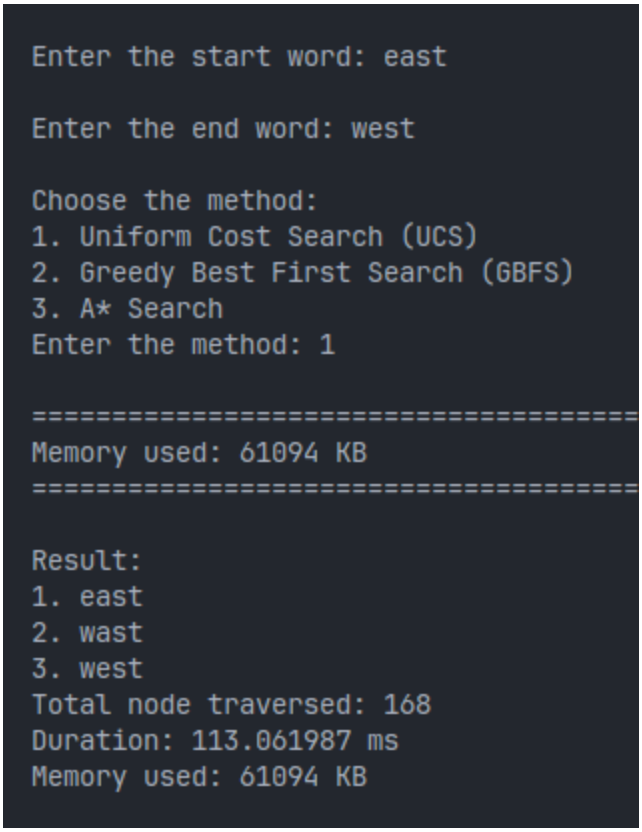
c. Algoritma A* Search

```
Enter the start word: expert

Enter the end word: cooler

Choose the method:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Enter the method: 3

====================================
Memory used: 73232 KB
====================================

Result:
1. expert
2. export
3. exhort
4. enhort
5. enfort
6. enfork
7. unfork
8. uncork
9. uncore
10. encore
11. endore
12. endere
13. enders
14. eiders
15. ciders
16. coders
17. cooers
18. cooees
19. cooeed
20. cooled
21. cooler
Total node traversed: 783
Duration: 1005.766188 ms
Memory used: 73232 KB
```

Gambar 4.6: Test case 2 Algoritma A* Search

4.3 Test Case 3

a. Algoritma UCS



Gambar 4.7: Test case 3 Algoritma UCS

b. Algoritma GBFS

```
Enter the start word: east

Enter the end word: west

Choose the method:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Enter the method: 2

=======================================
Memory used: 71074 KB
=======================================

Result:
1. east
2. wast
3. west
Total node traversed: 3
Duration: 40.002965 ms
Memory used: 71074 KB
```

Gambar 4.8: Test case 3 Algoritma GBFS

c. Algoritma A* Search

```
Enter the start word: east

Enter the end word: west

Choose the method:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Enter the method: 3

=====================================
Memory used: 61470 KB
=====================================

Result:
1. east
2. wast
3. west
Total node traversed: 3
Duration: 39.421752 ms
Memory used: 61470 KB
```

Gambar 4.9: Test case 3 Algoritma A* Search

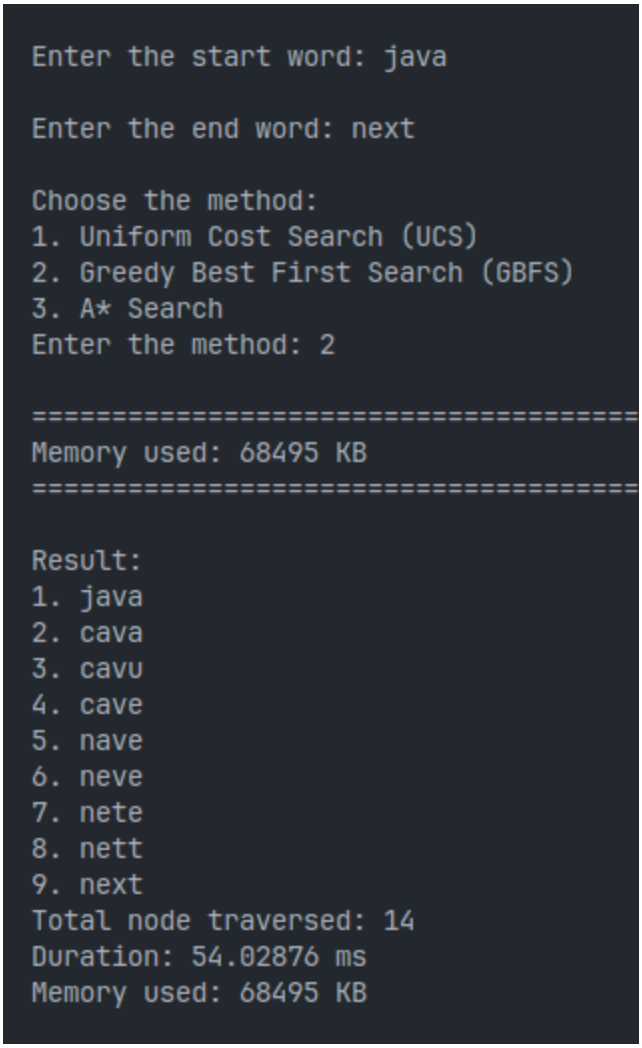
4.4 Test Case 4

a. Algoritma UCS

```
Enter the start word: java

Enter the end word: next

Choose the method:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Enter the method: 1

=====================================
Memory used: 71943 KB
=====================================

Result:
1. java
2. jara
3. vara
4. vera
5. vert
6. vext
7. next
Total node traversed: 6058
Duration: 925.027321 ms
Memory used: 71943 KB
```

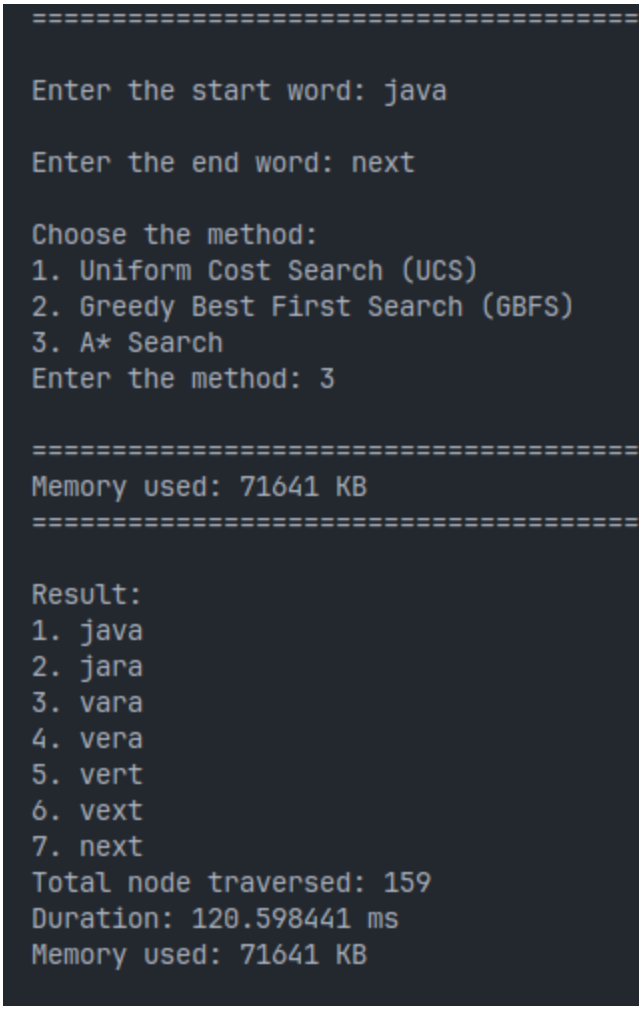
Gambar 4.10: Test case 4 Algoritma UCS

b. Algoritma GBFS



Gambar 4.11: Test case 4 Algoritma GBFS

c. Algoritma A* Search



Gambar 4.12: Test case 4 Algoritma A* Search

4.5 Test Case 5

a. Algoritma UCS

```
Enter the start word: vaccine

Enter the end word: iceberg

Choose the method:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Enter the method: 1

========================================
Memory used: 64277 KB
========================================

Result: No solution found!
Total node traversed: 2
Duration: 76.015177 ms
Memory used: 64277 KB
```

Gambar 4.13: Test case 5 Algoritma UCS

b. Algoritma GBFS

```
Enter the start word: vaccine

Enter the end word: iceberg

Choose the method:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Enter the method: 2

========================================
Memory used: 63270 KB
========================================

Result: No solution found!
Total node traversed: 2
Duration: 78.126397 ms
Memory used: 63270 KB
```

Gambar 4.14: Test case 5 Algoritma GBFS

c. Algoritma A* Search

```
Enter the start word: vaccine

Enter the end word: iceberg

Choose the method:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Enter the method: 3

========================================
Memory used: 67672 KB
========================================

Result: No solution found!
Total node traversed: 2
Duration: 52.546073 ms
Memory used: 67672 KB
```

Gambar 4.15: Test case 5 Algoritma A* Search

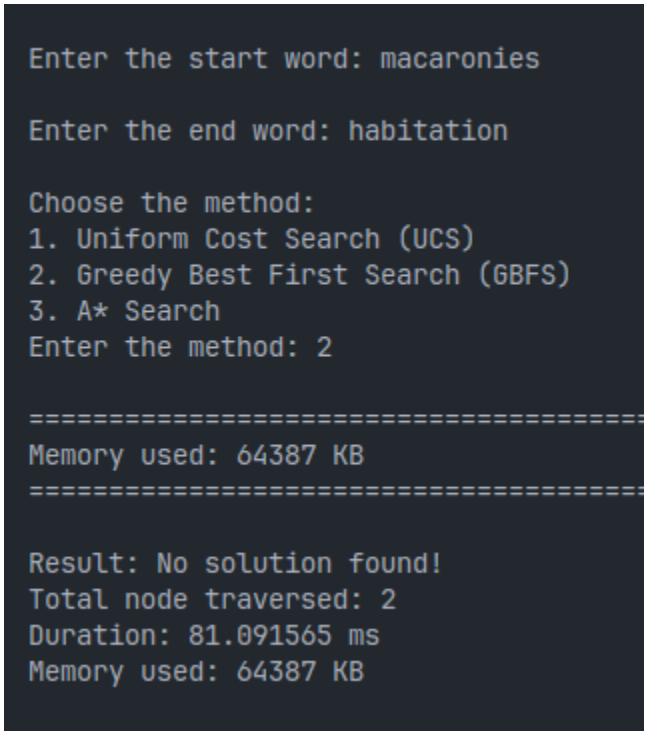
4.6 Test Case 6

a. Algoritma UCS

```
Enter the start word: macaronies

Enter the end word: habitation

Choose the method:
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search
Enter the method: 1

========================================
Memory used: 72401 KB
========================================

Result: No solution found!
Total node traversed: 2
Duration: 72.180445 ms
Memory used: 72401 KB
```

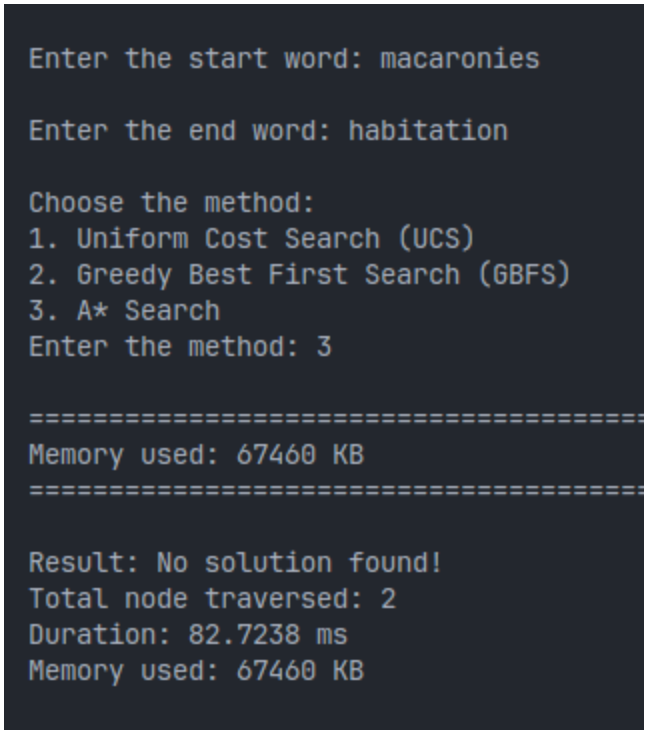
Gambar 4.16: Test case 6 Algoritma UCS

b. Algoritma GBFS



Gambar 4.17: Test case 6 Algoritma GBFS

c. Algoritma A* Search



Gambar 4.18: Test case 6 Algoritma A* Search

BAB 5: Analisis

Dari segi pencairan solusi yang optimal, Algoritma Uniform Cost Search (UCS) dan A* Search menghasilkan solusi yang optimal namun tidak untuk Algoritma Greedy Best First Search (GBFS). Hal ini dapat dilihat dari test case 2 (bab 4.2, expert -> cooler) dimana algoritma UCS dan A* search menghasilkan solusi dengan panjang 21 kata, sementara algoritma GBFS dengan panjang 52 kata. Begitu juga di test case 4 (bab 4.4, java -> next) dimana algoritma UCS dan A* search menghasilkan solusi dengan panjang 7 kata, sementara algoritma GBFS dengan panjang 9 kata.

Secara teoritis, time complexity untuk algoritma UCS, GBFS, dan A* search adalah `O(b^m)` (sama semua) Dimana b adalah branching factor (jumlah kata yang panjangnya sama pada kamus) dan m adalah depth dari tree (panjang langkah startword ke end word). Namun kompleksitas tidak memberikan informasi secara pasti algoritma mana yang seharusnya lebih cepat jika kompleksitas yang dibandingkan sama (karena worst case dan hanya memberikan informasi how fast it grows, bukan how much it grows). Dari data uji coba, test case 1, 2, dan 4 dimana simpul yang dikunjungi beragam (tidak dekat-dekat valuenya), dapat dilihat bahwa semakin banyak simpul yang dikunjungi akan semakin lama durasi perhitungannya. Dari ketiga uji coba tersebut, `n_gbfs < n a* < n_ucs` dan juga `t_gbfs < t_a* < t_ucs`. Untuk test case 3, jumlah node yang dikunjungi oleh algoritma A* search dan GBFS sama, akibatnya durasi perhitungannya juga akan mirip. Dari uji coba case 3, `n_gbfs ~ n a* < n_ucs` dan juga `t_gbfs ~ t_a* < t_ucs`. Untuk uji coba ke 5 dan ke 6, dimana solusi tidak ditemukan, dapat dilihat juga karena `n_gbfs ~ n a* ~ n_ucs` dan juga `t_gbfs ~ t_a* ~ t_ucs`. Artinya, untuk sebagian banyak kasus, berlaku `t_gbfs < t_a* < t_ucs` yaitu GBFS lebih cepat dari A*, dan A* lebih cepat dari UCS.

Untuk penggunaan memori, dapat dilihat dari semua test case bahwa tidak ada pola yang jelas atau umum untuk mendeskripsikan penggunaan memori. Penggunaan memorinya pun tidak terlalu ada signifikan perbedaannya (dibandingkan dengan durasi

waktu pencarian). Pada test case 1, berlaku `m_a* < m_gbfs < m_ucs`. Pada test case 2 berlaku `m_ucs < m_gbfs < m_a*`. Pada test case 3 berlaku `m_ucs < m_a* < m_gbfs`. Pada test case 4 berlaku `m_gbfs < m_ucs < m_a*`. Pada test case 5 berlaku `m_gbfs < m_ucs < m_a*`. Pada test case 6 berlaku `m_gbfs < m_a* < m_ucs`. Penggunaan memori juga diperngaruhi test case / nilai masukan yang diberikan.

BAB 6: Penutup

Kesimpulan

Permainan Word Ladder dapat diselesaikan dengan algoritma Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A* Search. Untuk mendapatkan solusi optimal namun sedikit lebih lama, gunakan algoritma UCS atau A* search. Untuk mendapatkan solusi yang paling cepat namun tidak optimal, gunakan algoritma GBFS.

Lampiran

1. Link Repository Github
   https://github.com/dewodt/Tucil3_13522011
2. Referensi dictionary
   https://github.com/dwyl/english-words
3. Tabel Poin

No	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		✓

Note: untuk poin 2, 4, 5 ada yang tidak ada solusinya, seperti contoh tes