

# **Laporan Tugas Kecil 3**

IF2211 Strategi Algoritma

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



Disusun oleh :

Muhammad Yusuf Rafi  
13522009  
K-01

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

**2023**

# BAB I

## DESKRIPSI MASALAH

### 1.1 Word Ladder

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

## **1.2 Spesifikasi Program**

Program dalam bahasa Java berbasis CLI (Command Line Interface) yang dapat menemukan solusi permainan word ladder menggunakan algoritma UCS, Greedy Best First Search, dan A\*. Kata-kata yang dapat dimasukkan harus berbahasa Inggris. Program diharapkan dapat menemukan solusi paling optimal untuk menyelesaikan permainan Word Ladder ini. Solusi dari tiap algoritma dijadikan sebagai analisis dalam penentuan solusi teroptimal.

### **1.2.1 Input**

Program menerima masukan:

1. Start word dan end word. Program harus bisa menangani berbagai panjang kata.
2. Pilihan algoritma yang digunakan (UCS, Greedy Best First Search, atau A\*).

### **1.2.2 Output**

Program menampilkan:

1. Path yang dihasilkan dari start word ke end word (cukup 1 path saja).
2. Banyaknya node yang dikunjungi .
3. Waktu eksekusi program.

### **1.2.3 Bonus**

Berikut adalah spesifikasi bonus dari program yang dibangun.

1. Program dapat berjalan dengan basis GUI (Graphical User Interface) (Untuk kakas GUI dibebaskan asalkan program algoritma UCS, Greedy Best First Search, dan A\* dalam bahasa Java).

## BAB II LANDASAN TEORI

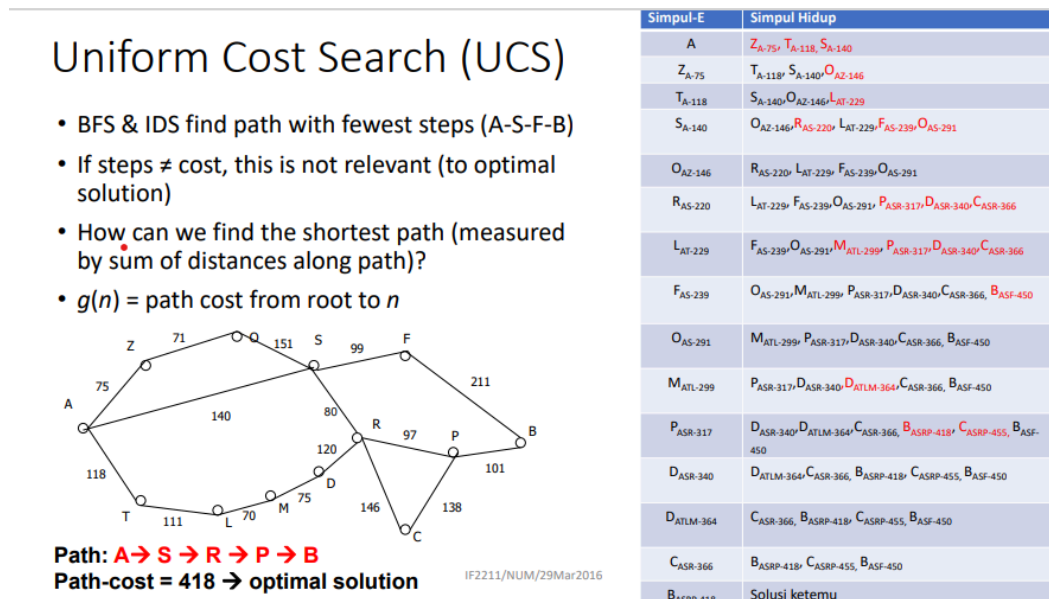
### 2.1 Algoritma Uniform Cost Search

Uniform Cost Search (UCS) adalah algoritma pencarian tanpa informasi yang bertujuan untuk menemukan rute terpendek. Algoritma ini menggabungkan prinsip dari Breadth-First Search (BFS) dan Iterative Deepening Search (IDS), dengan memberikan prioritas pada simpul dengan biaya kumulatif terendah. Proses pencarian dimulai dari simpul awal dan secara bertahap memperluas pencarian ke simpul tetangga berdasarkan biaya kumulatifnya. UCS menggunakan struktur data antrian prioritas untuk mengatur simpul yang akan dieksplorasi.

Secara ringkas, langkah-langkah algoritma UCS adalah sebagai berikut:

1. Masukkan simpul awal ke dalam antrian prioritas.
2. Ambil simpul dengan biaya kumulatif terendah dari antrian. Jika simpul tersebut merupakan simpul tujuan, tampilkan rute terpendek dan biaya totalnya, kemudian akhiri pencarian. Jika tidak, periksa apakah simpul tersebut telah dikunjungi sebelumnya.
3. Jika simpul belum dikunjungi, tambahkan semua simpul tetangga ke dalam antrian prioritas dengan biaya kumulatif yang sesuai.

Ilustrasi algoritma tersebut melalui gambar di bawah ini.



**Gambar 2. Ilustrasi UCS Algorithm.**

## 2.2 Algoritma Greedy Best-First Search

Algoritma Greedy Best-First Search (GBFS) adalah algoritma pencarian yang bersifat uninformed, yang bertujuan untuk menemukan solusi dengan menggunakan heuristik yang mengarahkan pencarian ke simpul yang dianggap paling menjanjikan. Algoritma ini sering digunakan dalam pencarian rute terpendek dan masalah optimasi lainnya. Pencarian dimulai dari simpul awal dan berlanjut ke simpul tetangga yang memiliki nilai heuristik terendah, yaitu yang dianggap paling dekat dengan tujuan. Proses ini berlanjut hingga simpul tujuan ditemukan atau tidak ada simpul yang tersisa untuk dieksplorasi.

Secara ringkas, langkah-langkah algoritma GBFS adalah sebagai berikut::

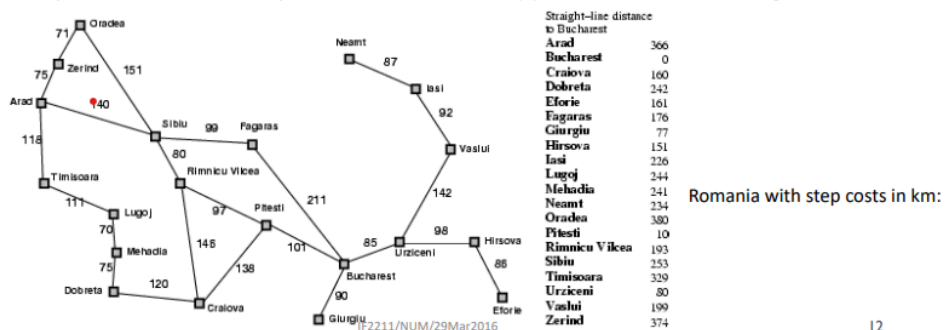
1. Masukkan simpul awal ke dalam daftar prioritas.
2. Ambil simpul dengan nilai heuristik terendah dari daftar prioritas. Jika simpul tersebut adalah simpul tujuan, tampilkan solusi dan akhiri pencarian. Jika tidak, periksa apakah simpul tersebut telah dikunjungi sebelumnya.
3. Jika simpul belum dikunjungi, tambahkan semua simpul tetangga ke dalam daftar prioritas dengan nilai heuristik yang sesuai.

Dengan pendekatan ini, GBFS cenderung menemukan solusi dengan cepat, tetapi tidak menjamin optimalitas karena keputusan pencarian didasarkan pada heuristik lokal tanpa mempertimbangkan gambaran keseluruhan dari masalah yang diselesaikan.

Ilustrasi algoritma tersebut melalui gambar di bawah ini.

### Greedy Best-First Search

- Idea: use an **evaluation function**  $f(n)$  for each node
  - $f(n) = h(n) \rightarrow$  estimates of cost from  $n$  to goal
  - e.g.,  $h_{SLD}(n)$  = straight-line distance from  $n$  to Bucharest
- Greedy best-first search expands the node that **appears** to be closest to goal



Gambar 3. Ilustrasi GBFS Algorithm.

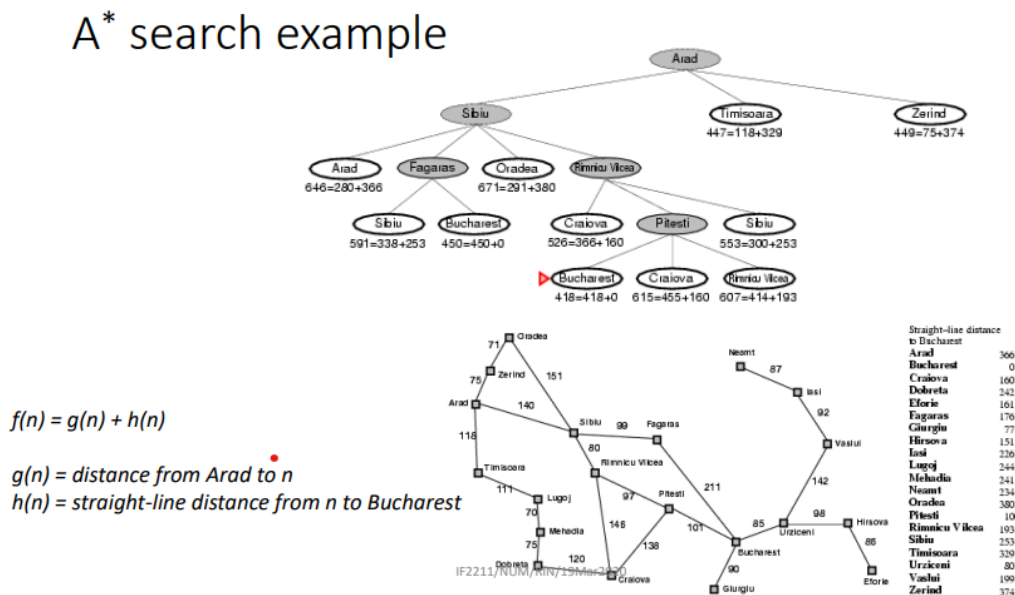
## 2.3 Algoritma A\*

Algoritma A\* adalah metode pencarian rute terpendek yang menggunakan pendekatan heuristik yang merupakan algoritma yang bersifat informed search. Secara konseptual, algoritma ini memperluas gagasan dari algoritma Uniform Cost Search (UCS). Tujuan utamanya adalah untuk mengurangi ekspansi jalur yang memiliki biaya tinggi atau mahal.

Dibandingkan dengan UCS yang hanya mempertimbangkan biaya aktual untuk mencapai suatu simpul, A\* mengambil pendekatan yang lebih canggih dengan memperhitungkan nilai estimasi heuristik. Fungsi biaya dalam A\* didefinisikan sebagai kombinasi dari biaya aktual ( $g(n)$ ) dan nilai estimasi heuristik ( $h(n)$ ), yaitu  $f(n) = g(n) + h(n)$ . Fungsi  $f(n)$  adalah cost sampai saat ini yang diperlukan untuk mencapai  $n$  dan Fungsi heuristik  $h(n)$  memberikan perkiraan biaya dari simpul  $n$  ke tujuan akhir.

Perbedaan A\* dari UCS adalah penggunaan nilai estimasi heuristik, yang memungkinkan algoritma ini untuk menghindari memperluas jalur-jalur yang cenderung lebih mahal. Penting untuk dicatat bahwa fungsi heuristik harus selalu memberikan perkiraan yang kurang dari atau sama dengan biaya sebenarnya (underestimate) agar A\* dapat menjamin pencarian rute yang optimal dan lengkap.

Ilustrasi algoritma tersebut melalui gambar di bawah ini.



Gambar 3. Ilustrasi GBFS Algorithm.

## BAB III IMPLEMENTASI

### 3.1 Implementasi Kelas Dasar

Berikut adalah daftar kelas yang dibutuhkan untuk implementasi sistem Word Ladder:

1. Node: Representasi simpul dalam graf, yang berisi kata, tetangga, biaya, dan referensi ke simpul induk.
2. Algorithm: Kelas abstrak yang digunakan sebagai dasar untuk algoritma pencarian rute.
3. Menu: Kelas yang menangani antarmuka pengguna dan logika aplikasi.
4. Main: Kelas utama untuk menjalankan program.
5. GUI: Kelas yang menangani antarmuka grafis pengguna dan logika aplikasi.

Dengan implementasi ini, Program memiliki seperangkat kelas yang mencakup struktur data dasar, algoritma pencarian rute, antarmuka pengguna, dan kelas utama untuk menjalankan program.

#### Node.java

```
import java.util.*;

public class Node {
    String word;
    List<Node> neighbors;
    int cost;
    Node parent;

    public Node(String word) {
        this.word = word;
        this.neighbors = new ArrayList<>();
        this.cost = 0;
        this.parent = null;
    }
}
```

#### Algorithm.java

```

import java.util.*;

public abstract class Algorithm {

    // Memeriksa apakah dua kata berbeda satu karakter
    static boolean isOneCharDiff(String word1, String word2) {
        int diffCount = 0;
        for (int i = 0; i < word1.length(); i++) {
            if (word1.charAt(i) != word2.charAt(i)) {
                diffCount++;
                if (diffCount > 1) return false;
            }
        }
        return true;
    }

    Object[] searchRoute(Node startNode, Node endNode, Set<String> dictionary,
        PriorityQueue<Node> queue) {
        Set<String> visited = new HashSet<>();
        int count = 0;
        long startTime = System.currentTimeMillis();

        while (!queue.isEmpty()) {
            count++;
            Node current = queue.poll();
            visited.add(current.word);

            if (current.word.equals(endNode.word)) {
                List<Node> path = new ArrayList<>();
                while (current != null) {
                    path.add(current);
                    current = current.parent;
                }
                Collections.reverse(path);
                long endTime = System.currentTimeMillis();
                long duration = endTime - startTime;
                return new Object[]{path, count, duration};
            }

            generateAdjacentWords(current, endNode, dictionary, visited, queue);
        }
    }
}

```



```

        long endTime = System.currentTimeMillis();
        long duration = endTime - startTime;
        return new Object[]{null, count, duration};
    }

    void generateAdjacentWords(Node current, Node target, Set<String>
dictionary, Set<String> visited, PriorityQueue<Node> queue) {
        int wordLength = current.word.length();

        for (int pos = 0; pos < wordLength; ++pos) {
            char origChar = current.word.charAt(pos);

            // Tukar current karakter dengan semua kemungkinan huruf alfabet
            for (char c = 'a'; c <= 'z'; ++c) {
                if (c == origChar) continue; // skip jika karakternya sama

                StringBuilder newWordBuilder = new StringBuilder(current.word);
                newWordBuilder.setCharAt(pos, c);
                String newWord = newWordBuilder.toString();

                // pastikan kata baru ada di dictionary dan belum dikunjungi
                if (dictionary.contains(newWord) && !visited.contains(newWord))
{
                    Node adjacent = new Node(newWord);
                    int newCost = calculateCost(newWord, current, target);
                    adjacent.cost = newCost;
                    adjacent.parent = current;
                    queue.add(adjacent);
                    visited.add(newWord);
                }
            }

            //Kembalikan Kata Sebelumnya
            current.word = current.word.substring(0, pos) + origChar +
current.word.substring(pos + 1);
        }
    }

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

```

```

        return diffCount;
    }

    abstract int calculateCost(String word, Node current, Node target);
}

```

## Menu.java

```

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class Menu {
    public static void run() {
        Scanner scanner = new Scanner(System.in);
        Set<String> dictionary = readDictionaryFromFile("src/word.txt");

        if(getInput(scanner) == 2){
            GUI gui = new GUI();
            gui.run();
            return;
        }

        while (true) {
            int choice = getMenuChoice(scanner);
            if (choice == 4) {
                System.out.println("Exiting...");
                break;
            }

            String startWord = getStartWord(scanner);
            String endWord = getEndWord(scanner);

            while (startWord.isEmpty() || endWord.isEmpty()) {
                System.out.println("Start word or end word cannot be empty.");
                startWord = getStartWord(scanner);
                endWord = getEndWord(scanner);
            }

            while(startWord.length() != endWord.length()) {

```

```

        System.out.println("Make Sure both words have the same length
:");

        startWord = getStartWord(scanner);
        endWord = getEndWord(scanner);
    }

    while(!dictionary.contains(startWord) ||
!dictionary.contains(endWord)) {
        System.out.println("Start word or end word not found in
dictionary. :");
        startWord = getStartWord(scanner);
        endWord = getEndWord(scanner);
    }

    Node startNode = new Node(startWord);
    Node endNode = new Node(endWord);

    PriorityQueue<Node> queue = new
PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
    queue.add(startNode);

    Algorithm algorithm;
    switch (choice) {
        case 1:
            algorithm = new UCS();
            break;
        case 2:
            algorithm = new GBFS();
            break;
        case 3:
            algorithm = new AStar();
            break;
        default:
            System.out.println("Invalid choice. Please try again.");
            continue;
    }

    System.out.println("\nSearching :) (Please Wait!!)");
    Object[] route = algorithm.searchRoute(startNode, endNode,
dictionary, queue);

    @SuppressWarnings("unchecked")
    List<Node> path = (List<Node>) route[0];
    int depth = (int) route[1];
    long duration = (long) route[2];

```

```

        // Menampilkan jalur transformasi
        System.out.println("\n===== " + (path != null ? "Path Found!!!":
"Path Not Found :(") + " =====");
        if (path != null) {
            System.out.print("Route: ");
            for (int i = 0; i < path.size(); i++) {
                System.out.print(path.get(i).word);
                if (i < path.size() - 1) {
                    System.out.print(" -> ");
                }
            }
        }
        System.out.println("\nNode Visited: " + depth);
        System.out.println("Execution time: " + duration + " milliseconds");

        if(Continue(scanner) != 1){
            break;
        }
    }

    scanner.close();
}

private static Integer getInput(Scanner scanner) {
    System.out.println("===== Pilih Metode Input =====");
    System.out.println("1. CLI ((Command Line Interface))");
    System.out.println("2. GUI (Graphical User Interface)");
    System.out.println("=====");

    System.out.print("Enter your choice: ");
    int choice = scanner.nextInt();
    scanner.nextLine();

    while(choice < 1 || choice > 4){
        System.out.println("Put the Right Input :)");
        System.out.print("Enter your choice: ");
        choice = scanner.nextInt();
        scanner.nextLine();
    }
    return choice;
}

```

```

private static int getMenuChoice(Scanner scanner) {
    System.out.println("\n===== Word Ladder Menu =====");
    System.out.println("1. Uniform Cost Search (UCS)");
    System.out.println("2. Greedy Best First Search (GBFS)");
    System.out.println("3. A* Search (AStar)");
    System.out.println("4. Exit");
    System.out.println("=====");

    System.out.print("Enter your choice: ");
    int choice = scanner.nextInt();
    scanner.nextLine();

    while(choice < 1 || choice > 4){
        System.out.println("Put the Right Input :)");
        System.out.print("Enter your choice: ");
        choice = scanner.nextInt();
        scanner.nextLine();
    }
    return choice;
}

private static String getStartWord(Scanner scanner) {
    System.out.print("Enter the start word: ");
    return scanner.nextLine().trim().toLowerCase();
}

private static String getEndWord(Scanner scanner) {
    System.out.print("Enter the end word: ");
    return scanner.nextLine().trim().toLowerCase();
}

private static Integer Continue(Scanner scanner){
    System.out.println("\n=====Pencarian Selesai!!!=====");
    System.out.println("1.Kembali ke Menu");
    System.out.println("2.Keluar");

    System.out.print("Pilihan : ");
    int input = scanner.nextInt();

    while (input < 1 || input > 2){
        System.out.println("\nMasukan Tidak Valid\n");
        System.out.print("Pilihan Menu : ");
    }
}

```

```

        input = scanner.nextInt();
    }

    return input;
}

// Metode untuk membaca kamus dari file
static Set<String> readDictionaryFromFile(String filePath) {
    Set<String> dictionary = new HashSet<>();
    try {
        BufferedReader reader = new BufferedReader(new FileReader(filePath));
        String line;
        while ((line = reader.readLine()) != null) {
            dictionary.add(line.trim()); // Menambahkan kata ke dalam kamus
        }
        reader.close();
    } catch (IOException e) {
        System.err.println("Error occurred while reading the dictionary file:
" + e.getMessage());
    }
    return dictionary;
}
}

```

### Main.java

```

public class Main {
    public static void main(String[] args) {
        Menu.run();
    }
}

```

### 3.2 Implementasi Kelas Turunan Algoritma

Berikut adalah daftar implementasi kelas turunan algoritma berdasarkan kelas Algorithm:

1. UCS : Override method calculateCost untuk menghitung biaya  $g(n)$ .
2. GBFS: Override method calculateCost untuk menghitung biaya  $h(n)$ .
3. A\*: Override method calculateCost untuk menghitung biaya  $f(n) = g(n) + h(n)$ .

Setiap kelas ini memperluas kelas abstrak `Algorithm` dan mengimplementasikan metode `calculateCost` sesuai dengan algoritma yang diterapkan. Dengan demikian, setiap algoritma memiliki cara yang berbeda dalam menghitung biaya rute.

Berikut, langkah-langkah algoritma UCS (Uniform Cost Search), GBFS (Greedy Best-First Search), dan A\* dalam program Word Ladder adalah sebagai berikut:

1. Inisialisasi node awal dan priority queue untuk menyimpan node yang akan diekspansi. Mulai dengan node awal yang merupakan kata awal yang diberikan *user* (Langkah ini di Menu.run).
2. Tambahkan node awal ke dalam priority queue.memprioritaskan ekspansi simpul berdasarkan biaya kumulatif (Langkah ini di Menu.run).
3. Selama priority queue tidak kosong, keluarkan node dengan biaya terendah dari antrian(elemen pertama) dan tandai sebagai telah dikunjungi (Langkah ini di Algorithm.SearchRoute).
4. Periksa apakah node yang dikeluarkan merupakan node tujuan yang ditentukan. Jika benar, maka pencarian selesai (Langkah ini di Algorithm.SearchRoute).
5. Jika tidak, ekspansi node tersebut dengan menambahkan semua tetangga yang belum dikunjungi ke dalam priority queue dengan biaya total yang telah dihitung(  $g(n)$  jika UCS,  $h(n)$  jika  $g(n)+h(n)$  jika A\*). Lakukan kembali langkah 3-5. (Langkah ini di Algorithm.generateAdjacentWords).
6. Jika pencarian selesai tanpa menemukan jalur dari node awal ke node tujuan, tampilkan pesan "Path Not Found :(".

#### 3.2.1 Implementasi Kelas Uniform Cost Search

**UCS.java**

```
public class UCS extends Algorithm {  
    @Override  
    int calculateCost(String word, Node current, Node target) {  
        return current.cost + 1;  
    }  
}
```

Berikut adalah langkah perhitungan biaya untuk  $g(n)$ :

1. Ambil biaya saat ini dari node yang sedang dievaluasi (`current.cost`).
2. Tambahkan dengan 1 ke biaya saat ini. Ini menunjukkan bahwa setiap langkah memiliki biaya tetap yang sama, yaitu 1.(simpul tetangga hanya boleh beda satu karakter).

### 3.2.2 Implementasi Kelas Greedy Best First Search

**GBFS.java**

```
public class GBFS extends Algorithm {  
  
    @Override  
    int calculateCost(String word, Node current, Node target) {  
        return OneCharDiff(word, target.word);  
    }  
}
```

Berikut adalah langkah perhitungan biaya untuk  $h(n)$ :

1. Hitung jumlah karakter yang berbeda antara kata saat ini dan kata tujuan dengan menghitung jumlah karakter yang berbeda antara dua kata dengan mengiterasi melalui setiap karakter dari kedua kata dan menambahkan 1 setiap kali karakter pada posisi yang sama tidak sama di antara kedua kata.  
(Perhitungan ini di Algorithm.OneCharDiff).

### 3.2.3 Implementasi Kelas A\*

**Astar.java**

```
public class AStar extends Algorithm {  
  
    @Override  
    int calculateCost(String word, Node current, Node target) {  
        return current.cost + 1 + OneCharDiff(word, target.word);  
    }  
}
```



Berikut adalah langkah perhitungan biaya untuk  $g(n) + h(n)$ :

1. Ambil biaya saat ini dari node yang sedang dievaluasi (`current.cost`).
2. Tambahkan dengan 1 ke biaya saat ini. Ini menunjukkan bahwa setiap langkah memiliki biaya tetap yang sama, yaitu 1.(simpul tetangga hanya boleh beda satu karakter).
3. Hitung jumlah karakter yang berbeda antara kata saat ini dan kata tujuan dengan menggunakan metode `OneCharDiff`.
4. Kembalikan nilai total biaya sebagai  $f(n)$ , yaki sebagai penjumlahan perhitungan biaya dengan UCS dan GBFS.

### 3.4 Implementasi GUI (Bonus)

#### GUI.java

```
import javax.swing.*.*;
import javax.swing.border.EmptyBorder;
import java.awt.*.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.*.*;

public class GUI {
    private JTextField startWordField;
    private JTextField endWordField;
    private JComboBox<String> algorithmComboBox;
    private JTextArea outputTextArea;

    public void run() {
        JFrame frame = new JFrame("Welcome to Word Ladder by Cupski");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(600, 400);
        frame.setLocationRelativeTo(null);

        JPanel mainPanel = new JPanel(new BorderLayout());
        mainPanel.setBorder(new EmptyBorder(10, 10, 10, 10));

        JPanel inputPanel = new JPanel(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.gridx = 0;
        gbc.gridy = 0;
        gbc.anchor = GridBagConstraints.WEST;
```

```

gbc.insets = new Insets(5, 5, 5, 5);

inputPanel.add(new JLabel("Start Word:"), gbc);
gbc.gridy++;
startWordField = new JTextField(15);
inputPanel.add(startWordField, gbc);

gbc.gridy++;
inputPanel.add(new JLabel("End Word:"), gbc);
gbc.gridy++;
endWordField = new JTextField(15);
inputPanel.add(endWordField, gbc);

gbc.gridy++;
inputPanel.add(new JLabel("Algorithm:"), gbc);
gbc.gridy++;
algorithmComboBox = new JComboBox<>(new String[]{"Uniform Cost Search (UCS)", "Greedy Best First Search (GBFS)", "A* Search (AStar)"});
inputPanel.add(algorithmComboBox, gbc);

gbc.gridy++;
gbc.gridwidth = 2;
JButton searchButton = new JButton("Search");
searchButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        searchButtonClicked();
    }
});
inputPanel.add(searchButton, gbc);

JPanel outputPanel = new JPanel(new BorderLayout());
outputPanel.setBorder(BorderFactory.createTitledBorder("Result"));

outputTextArea = new JTextArea();
outputTextArea.setEditable(false);
JScrollPane scrollPane = new JScrollPane(outputTextArea);
outputPanel.add(scrollPane, BorderLayout.CENTER);

mainPanel.add(inputPanel, BorderLayout.NORTH);
mainPanel.add(outputPanel, BorderLayout.CENTER);

frame.setContentPane(mainPanel);

frame.setVisible(true);
}

```

```

private void searchButtonClicked() {
    String startWord = startWordField.getText().trim().toLowerCase();
    String endWord = endWordField.getText().trim().toLowerCase();
    int algorithmChoice = algorithmComboBox.getSelectedIndex() + 1;

    Set<String> dictionary = Menu.readDictionaryFromFile("src/word.txt");

    // Validate input
    if (startWord.isEmpty() || endWord.isEmpty()) {
        JOptionPane.showMessageDialog(null, "Start word or end word cannot be empty.");
        return;
    }
    else if (!dictionary.contains(startWord) || !dictionary.contains(endWord)) {
        JOptionPane.showMessageDialog(null, "Start word or end word not found in dictionary.");
        return;
    }
    else if (startWord.length() != endWord.length()) {
        JOptionPane.showMessageDialog(null, "Make Sure both words have the same length");
        return;
    }

    // Run the search algorithm
    Node startNode = new Node(startWord);
    Node endNode = new Node(endWord);
    PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
    queue.add(startNode);
    Object[] route;
    switch (algorithmChoice) {
        case 1:
            Algorithm ucs = new UCS();
            route = ucs.searchRoute(startNode, endNode, dictionary, queue);
            break;
        case 2:
            Algorithm gbfs = new GBFS();
            route = gbfs.searchRoute(startNode, endNode, dictionary, queue);
            break;
        case 3:
            Algorithm aStar = new AStar();
            route = aStar.searchRoute(startNode, endNode, dictionary, queue);
            break;
    }
}

```

```

        default:
            JOptionPane.showMessageDialog(null, "Invalid algorithm choice.
Please try again.");
            return;
        }

        // Display the result
        displayResult(route);
    }

    private void displayResult(Object[] route) {
        @SuppressWarnings("unchecked")
        java.util.List<Node> path = (java.util.List<Node>) route[0];
        int depth = (int) route[1];
        long duration = (long) route[2];

        StringBuilder result = new StringBuilder();
        result.append(path != null ? "Path Found!!!\n" : "Path Not Found :(\n");
        if (path != null) {
            result.append("Route: \n");
            for (int i = 0; i < path.size(); i++) {
                result.append(i + 1).append(".
") .append(path.get(i).word).append("\n");
            }
        }
        result.append("\nNode Visited: ").append(depth);
        result.append("\nExecution time: ").append(duration).append("
milliseconds");

        outputTextArea.setText(result.toString());
    }
}

```

Kelas GUI ini menggunakan Java Swing untuk membuat antarmuka pengguna grafis (GUI) untuk aplikasi Word Ladder. GUI tersebut terdiri dari beberapa komponen yang mempermudah penggunaan aplikasi:

1. JTextField: Terdapat dua bidang teks, satu untuk memasukkan kata awal dan satu lagi untuk memasukkan kata akhir. Pengguna dapat memasukkan kata-kata yang ingin digunakan sebagai titik awal dan tujuan dalam pencarian Word Ladder.

2. JComboBox: Terdapat kotak kombinasi yang memungkinkan pengguna memilih algoritma pencarian yang ingin digunakan. Pilihan algoritma termasuk Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A\* Search (AStar). Pengguna dapat memilih algoritma yang sesuai dengan preferensi atau kebutuhan mereka.

3. JButton: Tombol "Search" memungkinkan pengguna memulai proses pencarian. Setelah pengguna memasukkan kata-kata awal dan akhir serta memilih algoritma pencarian, mereka dapat menekan tombol ini untuk memulai pencarian. Setelah proses pencarian selesai, hasilnya akan ditampilkan di area teks di bawahnya.

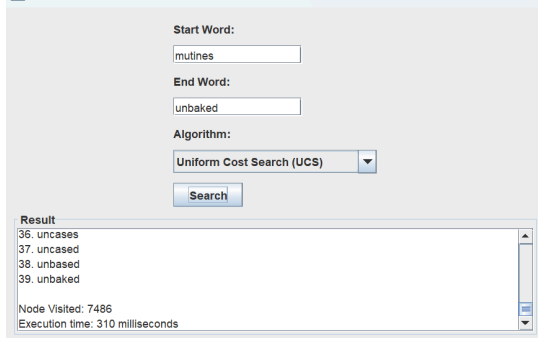
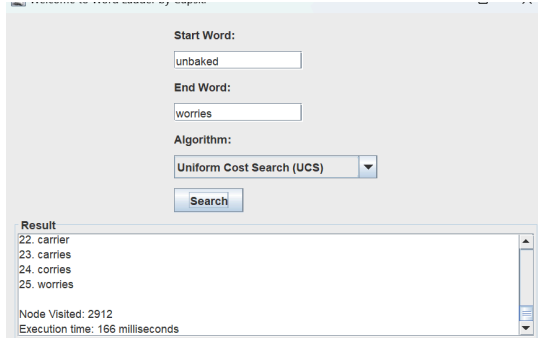
4. JTextArea: Area teks digunakan untuk menampilkan hasil pencarian. Setelah pencarian selesai, jalur yang ditemukan (jika ada), jumlah node yang dikunjungi, dan waktu eksekusi akan ditampilkan di area teks ini. Ini memberikan pengguna pemahaman yang jelas tentang hasil pencarian mereka.

Kombinasi komponen-komponen ini membuat penggunaan aplikasi Word Ladder menjadi lebih intuitif dan mudah dipahami.

### 3.4 Input dan Output Program UCS (3 dengan CLI, 3 dengan GUI)

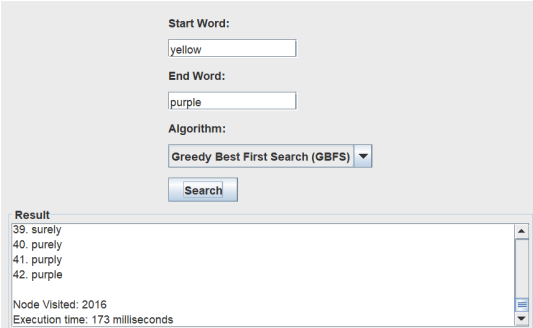
Input	Output
<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit ===== Enter your choice: 1 Enter the start word: poon Enter the end word: plea </pre>	<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit ===== Enter your choice: 1 Enter the start word: poon Enter the end word: plea  Searching :) (Please Wait!!)  ===== Path Found!!! ===== Route: poon -&gt; pood -&gt; plod -&gt; pled -&gt; plea Node Visited: 534 Execution time: 94 milliseconds  ===== Pencarian Selesai!!! ===== 1. Kembali ke Menu 2. Keluar Pilihan : █ </pre>

<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 1 Enter the start word: cage Enter the end word: zaps </pre>	<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 1 Enter the start word: cage Enter the end word: zaps  Searching :) (Please Wait!!)  ===== Path Found!!! ===== Route: cage -&gt; cape -&gt; caps -&gt; zaps Node Visited: 264 Execution time: 21 milliseconds  ====Pencarian Selesai!!!==== 1.Kembali ke Menu 2.Keluar </pre>
<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 1 Enter the start word: shops Enter the end word: works </pre>	<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 1 Enter the start word: shops Enter the end word: works  Searching :) (Please Wait!!)  ===== Path Found!!! ===== Route: shops -&gt; whops -&gt; woops -&gt; woods -&gt; words -&gt; works Node Visited: 1123 Execution time: 56 milliseconds  ====Pencarian Selesai!!!==== 1.Kembali ke Menu 2.Keluar </pre>
<pre> yellow purple Uniform Cost Search (UCS) </pre>	

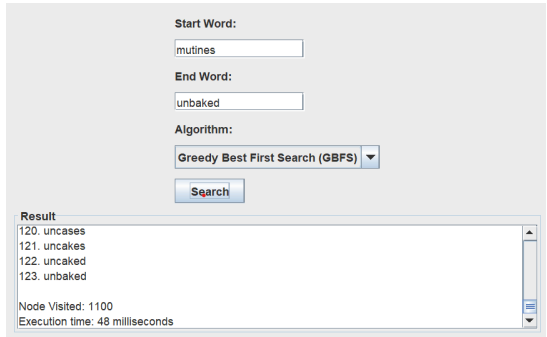
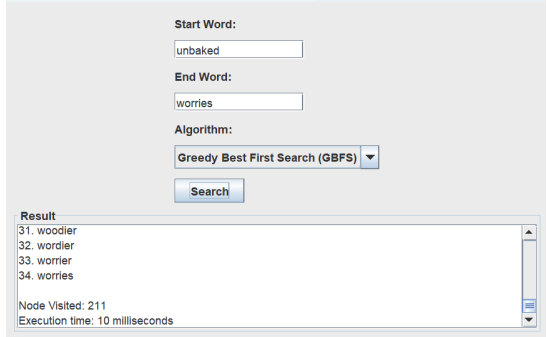
mutines unbaked Uniform Cost Search (UCS)	 <p>Start Word: mutines  End Word: unbaked  Algorithm: Uniform Cost Search (UCS)  Search</p> <p>Result  36. uncases  37. uncased  38. unbased  39. unbaked  Node Visited: 7486  Execution time: 310 milliseconds</p>
unbaked worries Uniform Cost Search (UCS)	 <p>Start Word: unbaked  End Word: worries  Algorithm: Uniform Cost Search (UCS)  Search</p> <p>Result  22. carrier  23. carries  24. corries  25. worries  Node Visited: 2912  Execution time: 166 milliseconds</p>

### 3.5 Input dan Output Program GBFS (3 dengan CLI, 3 dengan GUI)

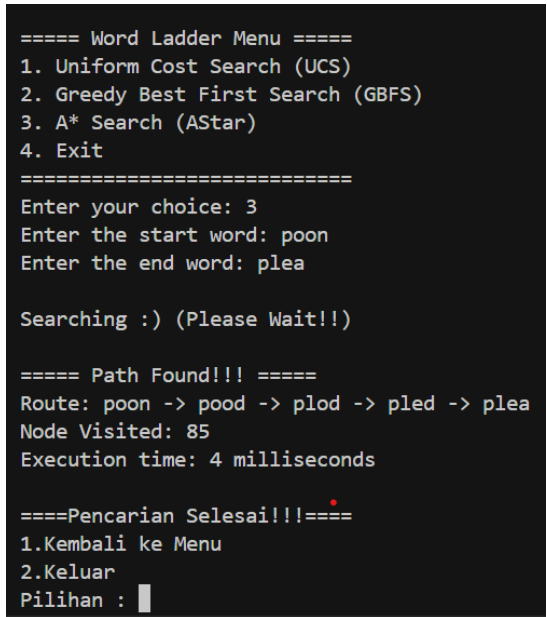
Input	Output
<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit ===== Enter your choice: 2 Enter the start word: poon Enter the end word: plea </pre>	<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit ===== Enter your choice: 2 Enter the start word: poon Enter the end word: plea  Searching :) (Please Wait!!)  ===== Path Found!!! ===== Route: poon -&gt; peon -&gt; peen -&gt; peed -&gt; pled -&gt; plea Node Visited: 6 Execution time: 0 milliseconds  =====Pencarian Selesai!!!===== 1.Kembali ke Menu 2.Keluar Pilihan : █ </pre>

<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 2 Enter the start word: cage Enter the end word: zaps </pre>	<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 2 Enter the start word: cage Enter the end word: zaps  Searching :) (Please Wait!!)  ===== Path Found!!! ===== Route: cage -&gt; cape -&gt; caps -&gt; zaps Node Visited: 4 Execution time: 2 milliseconds  ====Pencarian Selesai!!!==== 1.Kembali ke Menu 2.Keluar Pilihan : █ </pre>
<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 2 Enter the start word: shops Enter the end word: works </pre>	<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 2 Enter the start word: shops Enter the end word: works  Searching :) (Please Wait!!)  ===== Path Found!!! ===== Route: shops -&gt; whops -&gt; woops -&gt; woods -&gt; words -&gt; works Node Visited: 6 Execution time: 1 milliseconds  ====Pencarian Selesai!!!==== 1.Kembali ke Menu 2.Keluar Pilihan : █ </pre>
<pre> yellow purple Greedy Best First Search (GBFS) </pre>	

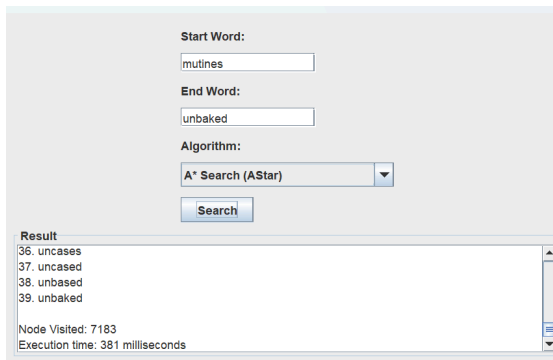
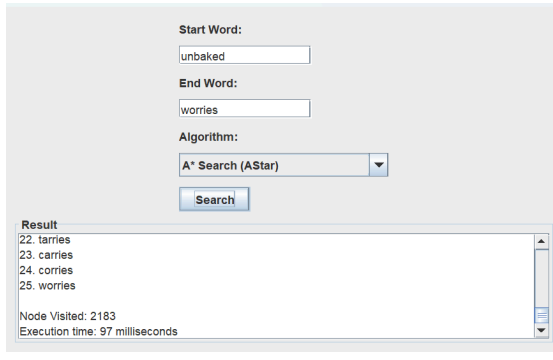


mutines unbaked Greedy Best First Search (GBFS)	
Unbaked worries Greedy Best First Search (GBFS)	

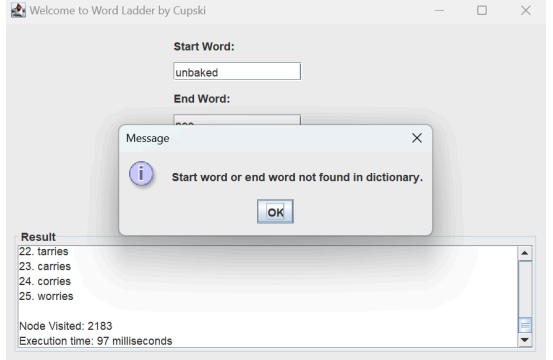
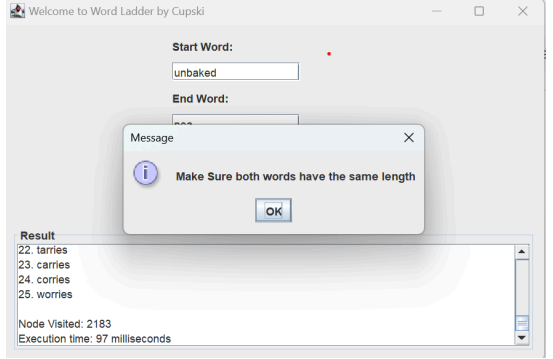
### 3.6 Input dan Output Program GBFS (3 dengan CLI, 3 dengan GUI)

Input	Output
<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 3 Enter the start word: poon Enter the end word: plea </pre>	

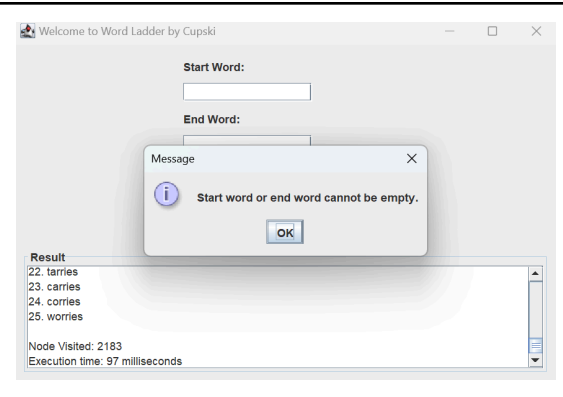
<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 3 Enter the start word: cage Enter the end word: zaps </pre>	<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 3 Enter the start word: cage Enter the end word: zaps  Searching :) (Please Wait!!)  ===== Path Found!!! ===== Route: cage -&gt; cape -&gt; caps -&gt; zaps Node Visited: 25 Execution time: 1 milliseconds  ====Pencarian Selesai!!!==== 1.Kembali ke Menu 2.Keluar Pilihan : █ </pre>
<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 3 Enter the start word: shops Enter the end word: works </pre>	<pre> ===== Word Ladder Menu ===== 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (AStar) 4. Exit =====  Enter your choice: 3 Enter the start word: shops Enter the end word: works  Searching :) (Please Wait!!)  ===== Path Found!!! ===== Route: shops -&gt; whops -&gt; woops -&gt; woods -&gt; words -&gt; works Node Visited: 101 Execution time: 4 milliseconds  ====Pencarian Selesai!!!==== 1.Kembali ke Menu 2.Keluar Pilihan : █ </pre>
<pre> yellow purple A* Search (AStar) </pre>	

mutines unbaked A* Search (AStar)	
unbaked worries A* Search (AStar)	

### 3.7 Validasi Input

Input	Output
unbaked poo A* Search (AStar)	
unbaked poo A* Search (AStar)	

## A\* Search (AStar)



### 3.8 Analisis Algoritma

Definisi dari  $f(n)$  sendiri dalam A\* adalah nilai total estimasi biaya dari node awal ke node tujuan melalui node  $n$  yang merupakan kombinasi antara biaya sejauh ini  $g(n)$  dan estimasi biaya yang tersisa  $h(n)$ . Maka dari itu, untuk 'CalculateCost' pada class AStar, saya programkan seperti pada 3.2.3. Adapun, untuk  $g(n)$ , sesuai yang telah dijelaskan sebelumnya adalah nilai biaya aktual atau sejauh ini dari node awal ke node  $n$ . Oleh Karena itu, untuk 'CalculateCost' pada class UCS, saya programkan seperti pada 3.2.1. Setiap Pergerakan node ditambahkan secara akumulatif dengan satu karena simpul tetangga yang mungkin hanya berbeda satu karakter, jadi saya tambahkan dengan satu.

Dalam kasus admissibility, penggunaan heuristik pada algoritma A\* adalah jumlah karakter yang berbeda antara kata saat ini dengan kata tujuan. Misalnya, jika kita sedang mempertimbangkan sebuah node yang merupakan kata dalam langkah pencarian, heuristik akan memberikan estimasi berapa banyak langkah lagi yang diperlukan untuk mencapai kata tujuan. Jika kita menetapkan biaya 1 untuk setiap langkah, heuristik ini akan memberikan nilai yang lebih rendah atau sama dengan biaya sebenarnya dari node tersebut ke tujuan, karena setiap langkah hanya bisa menghasilkan peningkatan maksimum satu unit pada nilai heuristik. Oleh karena itu, heuristik ini memenuhi kriteria admissibility, di mana  $h(n) \leq h^*(n)$ . Hal ini memberikan jaminan bahwa algoritma A\* akan mencapai solusi optimal atau solusi terpendek dalam konteks pencarian Word Ladder.

Secara konseptual, algoritma UCS (Uniform Cost Search) dan BFS (Breadth-First Search) memiliki tujuan yang sama, yaitu menemukan jalur terpendek dari node awal ke node tujuan. Namun, perbedaan teknis mendasar antara keduanya terletak pada cara mereka mengatur pengelolaan simpul-simpul yang dieksplorasi. Algoritma UCS menggunakan prinsip antrian prioritas, di mana simpul-simpul yang akan dieksplorasi selanjutnya dipilih berdasarkan biaya kumulatif mereka dari node awal. Dengan kata lain, simpul dengan biaya terendah dipilih untuk dieksplorasi terlebih dahulu. Sementara itu, BFS menggunakan prinsip antrian FIFO (First-In-First-Out), di mana simpul-simpul yang ditambahkan lebih awal ke antrian akan dieksplorasi lebih dulu.

Meskipun demikian, dalam konteks persoalan Word Ladder, di mana perhitungan biaya pada algoritma UCS bersifat akumulatif dan setiap langkah memiliki biaya tetap sebesar 1, secara logika prinsip kerja UCS menjadi mirip dengan BFS. Hal ini karena dalam kasus ini, biaya dari satu simpul ke simpul lainnya tidak bervariasi, sehingga prioritas ekspansi simpul bergantung pada urutan penambahan simpul ke antrian, mirip dengan prinsip FIFO yang digunakan oleh BFS sehingga akan menghasilkan jalur atau rute yang identik.

Dalam kasus Word Ladder, algoritma A\* diharapkan lebih efisien dibandingkan dengan UCS karena penggunaan heuristiknya yang mengarahkan pencarian dengan lebih efektif. Dalam Word Ladder, setiap transisi antara dua kata hanya dapat mengubah satu karakter, sehingga estimasi biaya menggunakan jumlah karakter yang berbeda antara kata saat ini dan kata tujuan (heuristik) sangat relevan. Juga algoritma A\* menggunakan fungsi  $f(n) = g(n) + h(n)$  yang berarti tidak hanya memperkirakan biaya simpul  $n$  saja, tapi juga ke simpul tujuan, serta algoritma A\* ini admissible sehingga diharapkan untuk lebih efisien dalam menemukan solusi optimal dibandingkan dengan UCS.

Terakhir, dalam kasus Word Ladder, heuristik yang digunakan oleh GBFS adalah jumlah karakter yang berbeda antara kata saat ini dan kata tujuan. Algoritma ini akan cenderung memilih simpul yang memiliki nilai heuristik yang lebih rendah, mengasumsikan bahwa jalur dengan sedikit perbedaan karakter lebih dekat dengan solusi yang diinginkan. Namun, karena GBFS hanya memperhatikan informasi lokal ini, tidak ada jaminan bahwa jalur yang dipilih akan menghasilkan solusi terpendek atau optimal. Algoritma ini dapat terjebak dalam memilih jalur yang tampaknya cepat menuju tujuan tetapi sebenarnya mengarah ke jalur yang lebih panjang atau tidak produktif. Dengan demikian, meskipun GBFS dapat memberikan solusi dalam waktu yang cepat, solusi yang dihasilkan tidak selalu optimal dalam hal panjang jalur atau jumlah langkah yang diperlukan untuk mencapai tujuan. Secara teoritis, GBFS tidak menjamin solusi optimal untuk persoalan word ladder.

### 3.9 Analisis dan Perbandingan Solusi

Dalam konteks optimalitas, UCS memberikan jaminan solusi optimal dalam persoalan Word Ladder karena secara eksplisit mengeksplorasi semua jalur dengan biaya kumulatif terendah, juga menghasilkan jalur yang identik dengan Algoritma BFS. Di sisi lain, Greedy GBFS tidak menjamin solusi optimal karena cenderung memilih jalur berdasarkan heuristik lokal, yaitu jumlah karakter yang berbeda antara kata saat ini dan kata tujuan. Meskipun GBFS cenderung lebih cepat daripada UCS dalam menemukan solusi, keputusan jalur yang hanya berdasarkan informasi lokal dapat mengakibatkan jalur yang lebih panjang dan tidak optimal. Sementara itu, A\* menawarkan solusi optimal dengan memanfaatkan heuristik yang admissible. Dengan menggunakan estimasi biaya yang lebih akurat, A\* dapat mengarahkan pencarian ke arah yang benar dan menemukan solusi dengan waktu eksekusi yang lebih efisien daripada UCS. Oleh karena itu, dari segi optimalitas, A\* adalah pilihan terbaik di antara ketiga algoritma ini untuk menyelesaikan persoalan Word Ladder. Sesuai dengan input dan output program pada 3.4, 3.5, 3.6, di mana A\* memberikan solusi optimal dengan efisien (walaupun, menghasilkan rute yang identik dengan UCS, tapi waktu eksekusinya lebih cepat).

Dalam konteks waktu eksekusi, UCS cenderung memerlukan waktu yang lebih lama karena mengeksplorasi semua kemungkinan secara melebar. Di sisi lain, GBFS seringkali lebih cepat karena fokusnya hanya pada langkah terbaik saat ini berdasarkan heuristik. Namun, kecepatannya terkadang dikompromikan oleh kemungkinan menghasilkan solusi yang tidak optimal karena sifatnya yang tidak terlalu memperhatikan biaya total. Sementara, A\* menunjukkan kinerja yang lebih cepat dibandingkan UCS karena mempertimbangkan juga biaya ke tujuan. Maka, A\* dapat secara efisien menemukan solusi optimal. Sesuai dengan input dan output program pada 3.4, 3.5, 3.6, di mana GBFS memiliki waktu eksekusi yang paling cepat, tapi solusinya tidak optimal.

Dalam konteks memori yang dibutuhkan, UCS cenderung memerlukan memori yang lebih besar karena harus menyimpan semua simpul yang dieksplorasi dalam antrian prioritas. Di sisi lain, GBFS cenderung memori yang lebih sedikit karena hanya menyimpan node yang paling menjanjikan, tetapi tidak menutup kemungkinan juga GBFS menghasilkan jalur yang lebih panjang karena selalu memilih yang dikira adalah jalur terbaik dan ternyata jalur tersebut tidak optimal. Sementara, A\* memerlukan memori yang cukup besar tapi lebih kecil daripada UCS karena penggunaan heuristik memungkinkan algoritma untuk fokus pada jalur-jalur yang lebih menjanjikan, mengurangi jumlah simpul yang perlu disimpan dalam memori. Oleh karena itu, A\* biasanya lebih efisien dalam penggunaan memori dibandingkan UCS dan seringkali bahkan lebih efisien daripada GBFS. Sesuai dengan input dan output program pada 3.4, 3.5, 3.6, di mana GBFS hanya melewati node yang lebih sedikit daripada UCS dan A\*, juga dengan A\* cenderung melewati node yang lebih sedikit daripada UCS

## BAB 4. Lampiran

### Lampiran 1

*Checklist penilaian :*

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa kesalahan	✓	
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	✓	

### Lampiran 2

Link Repository github : [https://github.com/cupski/Tucil3\\_13522009](https://github.com/cupski/Tucil3_13522009)