

# LAPORAN TUGAS KECIL 3

## IF2211 Strategi Algoritma

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



Disusun Oleh :  
Kharris Khisunica (13522051)

PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
2023

# Daftar Isi

Daftar Isi	2
1. Deskripsi Tugas	3
2. Analisis dan Implementasi Algoritma	4
2.1. Algoritma <i>Uniform Cost Search</i>	4
2.2. Algoritma <i>Greedy Best First Search</i>	4
2.3. Algoritma $A^*$	5
2.4. Analisis	6
3. Snippet Program	8
4. Test Case	24
5. Analisis Perbandingan	38
6. Implementasi Bonus	40
Lampiran	44

# 1. Deskripsi Tugas

*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.

## How To Play

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



Permainan ini akan dibuat dalam bahasa Java, dan menggunakan algoritma UCS, *Greedy Best First Search*, dan A\*. Kata-kata yang dapat dimasukkan juga harus dalam bahasa Inggris, yang akan divalidasi menggunakan kamus Collins Scrabble Words. Pengguna memasukkan kata awal dan kata akhir dan memilih algoritma yang digunakan, lalu program akan memberi luaran *Path* yang dihasilkan dari kata awal ke kata akhir, banyaknya *node* yang dikunjungi, serta waktu eksekusi program.

## 2. Analisis dan Implementasi Algoritma

### 2.1. Algoritma *Uniform Cost Search*

#### 2.1.1 Penjelasan

Uniform Cost Search adalah algoritma Search Tree (graph) yang digunakan untuk menyelesaikan beberapa persoalan. Algoritma ini memulai pencarian dari root node, kemudian dilanjutkan ke node-node selanjutnya. Dimana node tersebut dipilih yang memiliki harga (cost) terkecil dari root node. Algoritma ini merupakan modifikasi dari Bread First Search (BFS).

#### 2.1.2 Implementasi Algoritma

- a. Cek apakah kata sekarang sudah sama dengan kata akhir. Jika sudah sama, maka pencarian dihentikan dan rute dibentuk.
- b. Jika kata sekarang belum sama dengan kata akhir, maka bangkitkan pohon pencarian baru, dengan simpul bertetangga adalah seluruh variasi dari kata sekarang jika dirubah hanya satu huruf yang bukan kata sekarang dan ada di dalam kamus. Misal kata sekarang adalah "ABCD". Maka simpul-simpul yang dibangkitkan adalah "BBCD", "CBCD", ..., "ZBCD", "AACD", "ACCD", ... "AZCD", ..., "ABCZ" dan jika kata tersebut tidak ada di dalam kamus, maka simpul tersebut dibuang. Setelah pembangkitan, simpul baru akan menjadi simpul hidup dan kata sekarang tidak menjadi simpul hidup lagi.
- c. Kemudian, langkah a akan diulangi dengan simpul lain yang memiliki nilai  $g(n)$  terkecil, dimana  $g(n)$  adalah total kata yang telah dilewati dari kata awal ke kata sekarang.

### 2.2. Algoritma *Greedy Best First Search*

#### 2.2.1 Penjelasan

Algoritma Greedy Best-First Search adalah salah satu algoritma informed search di mana fungsi evaluasinya adalah fungsi heuristik. Algoritma ini memanfaatkan representasi graf untuk menyelesaikan permasalahannya. Tujuan utamanya adalah bergerak dari simpul awal menuju simpul tujuan dengan langkah yang paling optimal melalui simpul-simpul yang paling sesuai menurut fungsi heuristik yang didefinisikan.

#### 2.2.2 Implementasi Algoritma

- a. Cek apakah kata sekarang sudah sama dengan kata akhir. Jika sudah sama, maka pencarian dihentikan dan rute dibentuk.
- b. Jika kata sekarang belum sama dengan kata akhir, maka bangkitkan pohon pencarian baru, dengan simpul bertetangga adalah seluruh variasi dari kata sekarang jika dirubah hanya satu huruf yang bukan kata sekarang dan ada di dalam kamus. Misal kata sekarang adalah "ABCD". Maka simpul-simpul yang dibangkitkan adalah "BBCD", "CBCD", ..., "ZBCD", "AACD", "ACCD", ... "AZCD", ..., "ABCZ" dan jika kata tersebut tidak ada di dalam kamus, maka simpul tersebut dibuang. Setelah pembangkitan, simpul baru akan menjadi simpul hidup dan kata sekarang tidak menjadi simpul hidup lagi.
- c. Kemudian, langkah a akan diulangi dengan simpul lain yang memiliki nilai  $h(n)$  terkecil, dimana  $h(n)$  adalah total huruf berbeda antara kata sekarang dengan kata akhir.

## 2.3. Algoritma A\*

### 2.3.1 Penjelasan

Algoritma A\* adalah salah satu algoritma informed search di mana fungsi evaluasinya adalah fungsi heuristik dan fungsi cost. Algoritma ini memanfaatkan representasi graf untuk menyelesaikan permasalahannya. Tujuan utamanya adalah bergerak dari simpul awal menuju simpul tujuan dengan langkah yang paling optimal melalui simpul-simpul yang paling sesuai menurut fungsi yang didefinisikan.

### 2.3.2 Implementasi Algoritma

- a. Cek apakah kata sekarang sudah sama dengan kata akhir. Jika sudah sama, maka pencarian dihentikan dan rute dibentuk.
- b. Jika kata sekarang belum sama dengan kata akhir, maka bangkitkan pohon pencarian baru, dengan simpul bertetangga adalah seluruh variasi dari kata sekarang jika dirubah hanya satu huruf yang bukan kata sekarang dan ada di dalam kamus. Misal kata sekarang adalah "ABCD". Maka simpul-simpul yang dibangkitkan adalah "BBCD", "CBCD", ..., "ZBCD", "AACD", "ACCD", ... "AZCD", ..., "ABCZ" dan jika kata tersebut tidak ada di dalam kamus, maka simpul tersebut dibuang. Setelah pembangkitan, simpul baru akan menjadi simpul hidup dan kata sekarang tidak menjadi simpul hidup lagi.
- c. Kemudian, langkah a akan diulangi dengan simpul lain yang memiliki nilai  $g(n) + h(n)$  terkecil, dimana  $g(n)$  adalah total kata yang telah dilewati dari kata awal ke kata sekarang dan  $h(n)$  adalah total huruf berbeda antara kata sekarang dengan kata akhir.

## 2.4. Analisis

- a. Definisi dari  $f(n)$  dan  $g(n)$

$f(n)$  = total cost estimasi dari kata  $n$  sampai kata akhir  
 $g(n)$  = total cost sejauh ini untuk ke kata  $n$  = total kata yang sudah dilalui dari kata awal ke kata  $n$ .

- b. Apakah heuristik yang digunakan pada algoritma A\* *Admissible* ?

$h(n)$  = cost estimasi dari kata  $n$  sampai kata akhir = total huruf di kata  $n$  yang berbeda dengan kata akhir.  
Suatu fungsi heuristik dikatakan *admissible* jika fungsi tersebut tidak melebihi-lebihkan estimasi biaya untuk mencapai goal state pada saat ini.  
Dengan fungsi heuristik yang dipakai, heuristik tidak akan mengestimasi lebih dari langkah minimal yang dibutuhkan. Hal ini dikarenakan dalam *word ladder*, setiap langkah valid hanya boleh merubah satu huruf saja. Oleh karena itu, fungsi heuristik yang ada adalah heuristik yang *admissible* untuk kasus ini.

- c. Pada kasus *word ladder*, apakah algoritma UCS sama dengan BFS?

Jika kita perhatikan, nilai  $g(n)$  pada setiap node pada level yang sama akan bernilai sama pula, karena mereka semua memiliki jarak yang sama dengan node akar/kata awal. Oleh karena itu, setiap node pada level yang sama akan diperlakukan dengan sama juga dalam pengurutan pembangkitan node. Hal ini sama seperti BFS yang membangkitkan semua node pada satu level sebelum melanjutkan pembangkitan node ke level berikutnya.

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

Secara teoritis, algoritma A\* akan lebih efisien dibandingkan dengan algoritma UCS. Hal ini dikarenakan UCS bertindak seperti BFS, dimana algoritma ini akan mengecek semua simpul baru pada level yang sama. Sementara A\* memiliki fungsi heuristik yang akan memprioritaskan rute yang lebih dekat dengan solusi.

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

Secara teoritis, algoritma A\* akan lebih efisien dibandingkan dengan algoritma UCS. Hal ini dikarenakan UCS bertindak seperti BFS, dimana algoritma ini akan mengecek semua simpul baru pada level yang sama. Sementara A\* memiliki fungsi heuristik yang akan memprioritaskan rute yang lebih dekat dengan solusi.

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

Secara teoritis, algoritma *Greedy Best First Search* tidak menjamin solusi optimal untuk persoalan ini. Hal ini karena saat algoritma ini memilih satu node dan menemukan node yang memiliki nilai  $h(n)$  yang lebih rendah, maka node lain dengan  $h(n)$  yang lebih tinggi akan diabaikan. Hal ini bisa menjadi masalah karena sering kali terdapat banyak node dengan  $h(n)$  yang sama tapi bisa menghasilkan rute dengan panjang yang berbeda.

### 3. Snippet Program

## Main.Java (CLI)

[illegible]



```
public static boolean getRestartChoice(Scanner scanner){
    String choice;

    while (true){
        System.out.print(s:"Do you want to restart the program? (y/n): ");
        choice = scanner.next();

        if(choice.equalsIgnoreCase(anotherString:"y")){
            return true;
        }
        else if (choice.equalsIgnoreCase(anotherString:"n")){
            return false;
        }
        else{
            System.out.println(x:"Invalid choice. Please choose 'y' or 'n' ");
        }
    }
}

public static void clearScreen() {
    String os = System.getProperty(key:"os.name").toLowerCase();
    ProcessBuilder processBuilder = new ProcessBuilder();
    try {
        if (os.contains(s:"win")) {
            processBuilder.command(...command:"cmd", "/c", "cls");
        }
        Process process = processBuilder.inheritIO().start();
        process.waitFor();
    } catch (IOException | InterruptedException e) {
        System.out.println("Error clearing screen: " + e.getMessage());
    }
}
```

Run | Debug

```
public static void main(String[] args){

    String filename = "src/dictionary.txt";
    Scanner scanner = new Scanner(System.in);
    DictLoader dictLoader = new DictLoader(filename);
    Set<String> dict = dictLoader.getDict();

    boolean restart;

    do {
        greeting();
        System.out.println(x:"=====");
        List<String> inputWord = WordLadderUtils.readInput(scanner, dict);
        String start = inputWord.get(index:0);
        String goal = inputWord.get(index:1);

        int choice = algorithmOpt(scanner);
        System.out.println();
        System.out.println(x:"=====");

        switch (choice) {
            case 1:
                WordLadderUtils.WordLadderResult ladder = UCS.ucsWordLadder(start, goal, dict);

                if (ladder != null){
                    UCS
                    System.out.println(x:"UCS Word Ladder: " );
                    WordLadderUtils.printPath(ladder.getWordLadderPath());
                    System.out.println();
                    System.out.println("UCS Visited Nodes Amount: " + ladder.getVisitedNum());
                    System.out.println("Time: " + ladder.getExecTime() + "ms");
                }
            else{
                System.out.println["Womp Womp. There's no possible solution from "
                    + start + " to " + goal + " o(T-To)"];
            }
        }
    } while (restart);
}
```

```

case 2:

ladder = GBFS.gbfsWordLadder(start, goal, dict);

if (ladder != null){
    System.out.println(x:"GBFS Word Ladder: ");
    WordLadderUtils.printPath(ladder.getWordLadderPath());
    System.out.println();
    System.out.println("GBFS' Visited Nodes Amount: " + ladder.getVisitedNum());
    System.out.println("Time: " + ladder.getExecTime() + "ms");
}
else{
    System.out.println("Womp Womp. There's no possible solution from " + start
        + " to " + goal + " o(T-To)");
}
break;

case 3:

ladder = AStar.aStarWordLadder(start, goal, dict);
if (ladder != null){
    System.out.println(x:"A* Word Ladder: ");
    WordLadderUtils.printPath(ladder.getWordLadderPath());
    System.out.println();
    System.out.println("A* Visited Nodes Amount: " + ladder.getVisitedNum());
    System.out.println("Time: " + ladder.getExecTime() + "ms");
}
else{
    System.out.println("Womp Womp. There's no possible solution from " + start
        + " to " + goal + " o(T-To)");
}
break;

```

```

case 4:
    exitGreeting();
    scanner.close();
    return;

default:
    break;
}

restart = getRestartChoice(scanner);

}while (restart);

scanner.close();

```

DictLoader.java

```
import java.util.HashSet;
import java.util.Set;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class DictLoader {
    private Set<String> dict;

    public DictLoader(String filename){
        dict = new HashSet<>();
        readDict(filename);
    }

    public void readDict(String filename){
        File myFile = new File(filename);
        Scanner scanner = null;
        try{
            scanner = new Scanner(myFile);
            while (scanner.hasNextLine()){
                String word = scanner.nextLine().trim();
                dict.add(word);
            }
        } catch (FileNotFoundException e){
            System.out.println(x:"File Not Found. Please Try Again!");
            e.printStackTrace();
        } finally{
            if (scanner != null){
                scanner.close();
            }
        }
    }

    public Set<String> getDict(){
        return dict;
    }
}
```

### WordNode.java

```
public class WordNode {
    String word;
    WordNode parent;
    int cost;
    int heuristic;

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

### WordLadderUtils.java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Scanner;
import java.util.Set;
import java.util.List;

class WordLadderUtils{
    public static List<String> readInput(Scanner scanner, Set<String> dict){
        String start;
        String goal;
        while(true){
            System.out.print(s:"Please input the Start word: ");
            start = scanner.nextLine().toUpperCase().trim();
            System.out.print(s:"Please input the Goal Word: ");
            goal = scanner.nextLine().toUpperCase().trim();

            if (isInDict(start, dict) && isInDict(goal,dict) && start.length() == goal.length()){
                break;
            } else{
                System.out.println(x:"Ups. It seems like your words are either not the same length or");
                System.out.println();
            }
        }
        return Arrays.asList(start,goal);
    }

    public static boolean isInDict(String word, Set<String> dict){
        return dict.contains(word);
    }
}
```

```

public static int getHN(String word_now, String word_goal){
    int mismatched = 0;

    for (int i=0; i<word_now.length(); i++){
        if(word_now.charAt(i) != word_goal.charAt(i)){
            mismatched++;
        }
    }
    return mismatched;
}

public static List<String> getAdjWords(String word, Set<String> dict){
    List<String> adjWords = new ArrayList<>();
    char[] wordChars = word.toCharArray();

    for (int i=0; i<wordChars.length; i++){
        char oriChar = wordChars[i];
        for (char c = 'A'; c<='Z'; c++){
            if (c != oriChar){
                wordChars[i] = c;
                String newWord = new String(wordChars);
                if (dict.contains(newWord)){
                    adjWords.add(newWord);
                }
            }
        }
        wordChars[i] = oriChar;
    }
    return adjWords;
}

```

```

public static List<String> constructPath(WordNode node){
    List<String> path = new ArrayList<>();
    while (node != null){
        path.add(index:0, node.word);
        node = node.parent;
    }

    return path;
}

```

```
public static String highlightDiff(String word1, String goal){
    StringBuilder result = new StringBuilder();
    String Green = "\u001B[32m";
    String Reset = "\u001B[0m";

    for (int i=0; i<word1.length(); i++){
        if (word1.charAt(i) == goal.charAt(i)){
            result.append(Green).append(word1.charAt(i)).append(Reset);
        }
        else{
            result.append(word1.charAt(i));
        }
    }

    return result.toString();
}

public static void printPath(List<String> wlPath){
    String goal = wlPath.get(wlPath.size()-1);
    for (int i=0; i<wlPath.size(); i++){
        String wordNow = wlPath.get(i);

        if(i == wlPath.size()-1){
            System.out.print(highlightDiff(goal, goal));
        }
        else{
            System.out.print(highlightDiff(wordNow, goal) + "->");
        }
    }
}
```

```
public static class WordLadderResult{
    private List<String> wordLadderPath;
    private int visitedNum;
    private long exectime;

    public WordLadderResult(List<String> wordLadderPath, int visitedNum, long exectime){
        this.wordLadderPath = wordLadderPath;
        this.visitedNum = visitedNum;
        this.exectime = exectime;
    }

    public List<String> getWordLadderPath(){
        return this.wordLadderPath;
    }

    public int getVisitedNum(){
        return this.visitedNum;
    }

    public long getExecTime(){
        return this.exectime;
    }
}
```



## UCS.java

```
import java.util.*;

public class UCS {
    public static WordLadderUtils.WordLadderResult ucsWordLadder
        (String start, String goal, Set<String> dict){

        PriorityQueue<WordNode> queue = new PriorityQueue<>
            (Comparator.comparingInt(WordNode->WordNode.cost));
        Set<String> visited = new HashSet<>();
        queue.add(new WordNode(start, parent:null, cost:0, heuristic:0));
        visited.add(start);
        int count = 0;
        long startTime = System.nanoTime();
        while(!queue.isEmpty()){
            WordNode current = queue.poll();
            String currentWord = current.word;
            int currentCost = current.cost;
            count++;

            if (currentWord.equals(goal)){
                long endTime = System.nanoTime();
                List<String> wlPath = WordLadderUtils.constructPath(current);
                long exectime = (endTime-startTime)/1000000;
                return new WordLadderUtils.WordLadderResult(wlPath, count,exectime);
            }

            for (String Adj : WordLadderUtils.getAdjWords(currentWord, dict)){
                if (!visited.contains(Adj)){
                    queue.add(new WordNode(Adj, current, currentCost + 1,heuristic:0));
                    visited.add(Adj);
                }
            }
        }
        long endTime = System.nanoTime();
        long exectime = (endTime-startTime)/1000000;
        return new WordLadderUtils.WordLadderResult(wordLadderPath:null, count,exectime);
    }
}
```

## GBFS.java

```
import java.util.*;
public class GBFS {
    public static WordLadderUtils.WordLadderResult gbfsWordLadder
        (String start, String goal, Set<String> dict){

        PriorityQueue<WordNode> queue = new PriorityQueue<>
            (Comparator.comparingInt(WordNode->WordNode.heuristic));
        Set<String> visited = new HashSet<>();
        int HN = WordLadderUtils.getHN(start, goal);
        queue.add(new WordNode(start, parent:null, cost:0,HN));
        visited.add(start);
        int count = 0;
        long startTime = System.nanoTime();

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

            if (currentWord.equals(goal)){
                long endTime = System.nanoTime();
                List<String> wlPath = WordLadderUtils.constructPath(current);
                long exectime = (endTime-startTime)/1000000;
                return new WordLadderUtils.WordLadderResult(wlPath, count,exectime);
            }

            for (String Adj : WordLadderUtils.getAdjWords(currentWord, dict)){
                if (!visited.contains(Adj)){
                    int adjHN = WordLadderUtils.getHN(Adj, goal);
                    queue.add(new WordNode(Adj, current, cost:0,adjHN));
                    visited.add(Adj);
                }
            }
        }

        long endTime = System.nanoTime();
        long exectime = (endTime-startTime)/1000000;
        return new WordLadderUtils.WordLadderResult(wordLadderPath:null, count,exectime);
    }
}
```

## AStar.java

```
import java.util.*;
public class AStar {
    public static WordLadderUtils.WordLadderResult aStarWordLadder
        (String start, String goal, Set<String> dict){

        PriorityQueue<WordNode> queue =
            new PriorityQueue<>(Comparator.comparingInt(WordNode->WordNode.cost + WordNode.heuristic));
        Set<String> visited = new HashSet<>();
        int HN = WordLadderUtils.getHN(start, goal);
        queue.add(new WordNode(start, parent:null, cost:0, HN));
        visited.add(start);
        int count = 0;
        long startTime = System.nanoTime();

        while(!queue.isEmpty()){
            WordNode current = queue.poll();
            String currentWord = current.word;
            int currentCost = current.cost;
            count++;

            if (currentWord.equals(goal)){
                long endTime = System.nanoTime();
                List<String> wlPath = WordLadderUtils.constructPath(current);
                long exectime = (endTime-startTime)/1000000;

                return new WordLadderUtils.WordLadderResult(wlPath, count,exectime);
            }
            for (String Adj : WordLadderUtils.getAdjWords(currentWord, dict)){
                if (!visited.contains(Adj)){
                    int adjHN = WordLadderUtils.getHN(Adj, goal);
                    queue.add(new WordNode(Adj, current, currentCost+1, adjHN+1));
                    visited.add(Adj);
                }
            }
        }
        long endTime = System.nanoTime();
        long exectime = (endTime-startTime)/1000000;
        return new WordLadderUtils.WordLadderResult(wordLadderPath:null, count,exectime);
    }
}
```

## BONUS WordLadderGUI.java

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.EmptyBorder;
import javax.swing.table.*;
import java.util.*;

public class WordLadderGUI {
    private JFrame frame;
    private JComboBox<String> algorithmComboBox;
    private JTextField startNodeField, goalNodeField;
    private JButton searchButton;
    private JTextArea infoArea;
    private JPanel resultPanel;

    public WordLadderGUI() {
        frame = new JFrame(title:"Word Ladder Solver");
        frame.setSize(width:400, height:500);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());

        setupInputAndButton();
        setupResultPanel();
        setupInfoArea();

        frame.setVisible(true);
    }
}
```

```

private void setupInputAndButton() {
    // Setup Input Panel
    JPanel inputPanel = new JPanel(new BorderLayout());
    inputPanel.setBorder(new EmptyBorder(top:5, left:5, bottom:5, right:5));

    // Setup Fields Panel
    JPanel fieldsPanel = new JPanel(new GridLayout(rows:3, cols:2));
    fieldsPanel.setBorder(new EmptyBorder(top:5, left:5, bottom:5, right:5));
    algorithmComboBox = new JComboBox<>(new String[]{"UCS", "GBFS", "A*"});
    startNodeField = new JTextField();
    goalNodeField = new JTextField();

    fieldsPanel.add(new JLabel(text:"Start Word: "));
    fieldsPanel.add(startNodeField);
    fieldsPanel.add(new JLabel(text:"Goal's Word: "));
    fieldsPanel.add(goalNodeField);
    fieldsPanel.add(new JLabel(text:"Choose Algorithm: "));
    fieldsPanel.add(algorithmComboBox);

    searchButton = new JButton(text:"Start Searching!");
    searchButton.setPreferredSize(new Dimension(width:50, height:30));
    searchButton.addActionListener(e -> wlSearch());

    inputPanel.add(fieldsPanel, BorderLayout.CENTER);
    inputPanel.add(searchButton, BorderLayout.SOUTH);

    frame.add(inputPanel, BorderLayout.NORTH);
}

```

```

public void setupInfoArea() {
    infoArea = new JTextArea(rows:3, columns:20);
    infoArea.setEditable(b:false);
    frame.add(new JScrollPane(infoArea), BorderLayout.SOUTH);
}

```

```

public void setupResultPanel() {
    resultPanel = new JPanel();
    resultPanel.setLayout(new BorderLayout());
    frame.add(resultPanel, BorderLayout.CENTER);
}

```

```

public void w1Search() {
    String startNode = startNodeField.getText().toUpperCase().trim();
    String goalNode = goalNodeField.getText().toUpperCase().trim();
    String filename = "src/dictionary.txt";
    DictLoader dictLoader = new DictLoader(filename);
    Set<String> dict = dictLoader.getDict();

    if (startNode.isEmpty() || goalNode.isEmpty()) {
        JOptionPane.showMessageDialog(frame, message: "Please enter both start and goal word.");
        return;
    }

    if (startNode.length() != goalNode.length()) {
        JOptionPane.showMessageDialog(frame, message: "Error: Start and goal word must be of the same length.");
        return;
    }

    if (!WordLadderUtils.isInDict(startNode, dict) && !WordLadderUtils.isInDict(goalNode, dict)) {
        JOptionPane.showMessageDialog(frame, message: "Please enter valid words.");
        return;
    }

    WordLadderUtils.WordLadderResult result = null;
    switch ((String) algorithmComboBox.getSelectedItem()) {
        case "UCS" -> result = UCS.ucsWordLadder(startNode, goalNode, dict);
        case "GBFS" -> result = GBFS.gbfsWordLadder(startNode, goalNode, dict);
        case "A*" -> result = AStar.aStarWordLadder(startNode, goalNode, dict);
        default -> JOptionPane.showMessageDialog(frame, message: "Algorithm not implemented yet");
    }

    resultPanel.removeAll();

    if (result != null && result.getWordLadderPath() != null) {
        WLTable wlTable = new WLTable(result.getWordLadderPath());
        JTable table = wlTable.createTable();
        resultPanel.add(new JScrollPane(table), BorderLayout.CENTER);
        infoArea.setText("Path Length: " + result.getWordLadderPath().size() + "\nVisited Node's Amount: "
            + result.getVisitedNum() + "\nExecution Time: " + result.getExecTime() + "\n");
    } else {
        JTextArea msgArea = new JTextArea("No path found between " + startNode + " and " + goalNode);
        resultPanel.add(new JScrollPane(msgArea), BorderLayout.CENTER);
        infoArea.setText("Visited Node's Amount: " + result.getVisitedNum() + "\nExecution Time: "
            + result.getExecTime() + "\n");
    }

    frame.revalidate();
    frame.repaint();
}

```

```

public static class WTable {
    private String[][] wTableData;

    public WTable(java.util.List<String> wPath) {
        wTableData = new String[wPath.size()][1];
        for (int i = 0; i < wPath.size(); i++) {
            wTableData[i][0] = wPath.get(i);
        }
    }

    public JTable createTable() {
        int size = wTableData.length;
        String goal = wTableData[size-1][0];
        String[] columnNames = {size + "-Step Solution"};
        JTable table = new JTable(wTableData, columnNames);

        DefaultTableCellRenderer renderer = new DefaultTableCellRenderer() {
            @Override
            public Component getTableCellRendererComponent(JTable table, Object value,
                boolean isSelected, boolean hasFocus, int row, int column) {
                JLabel label = (JLabel) super.getTableCellRendererComponent(
                    table, value, isSelected, hasFocus, row, column);
                String word = (String) value;
                label.setText(highlightDiff(word, goal));
                label.setHorizontalAlignment(SwingConstants.CENTER);
                return label;
            }
        };
        table.getColumnModel().getColumn(columnIndex:0).setCellRenderer(renderer);

        table.setRowHeight(rowHeight:30);
        table.setFont(new Font(name:"Arial", Font.BOLD, size:20));
        return table;
    }
}

```

```

public static String highlightDiff(String word1, String goal){
    StringBuilder result = new StringBuilder(str:"<html>");
    for (int i=0; i<word1.length(); i++){
        if (word1.charAt(i) == goal.charAt(i)){
            result.append(str:"<font color='green'>").append(word1.charAt(i)).append(str:"</font>");
        }
        else{
            result.append(word1.charAt(i));
        }
    }
    result.append(str:"</html>");
    return result.toString();
}

```

```

Run | Debug
public static void main(String[] args) {
    SwingUtilities.invokeLater(new WordLadderGUI()); // Corrected class name
}
}

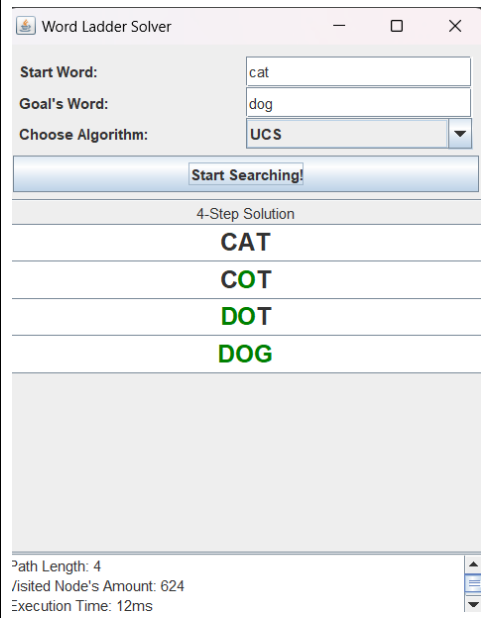
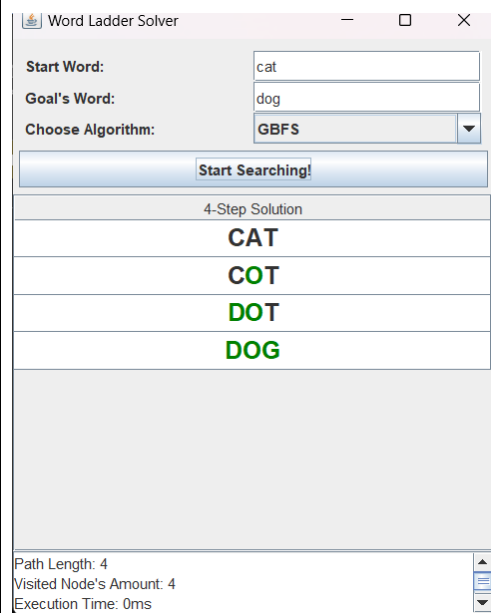
```

## 4. Test Case

Input ke-1

Kata Awal	cat
Kata Akhir	dog

Output

UCS	 <p>The screenshot shows the 'Word Ladder Solver' application window. The 'Start Word' is 'cat' and the 'Goal's Word' is 'dog'. The 'Choose Algorithm' dropdown is set to 'UCS'. A 'Start Searching!' button is visible. Below it, the '4-Step Solution' is displayed: CAT, COT, DOT, and DOG. At the bottom, the status bar shows 'Path Length: 4', 'Visited Node's Amount: 624', and 'Execution Time: 12ms'.</p>
GBFS	 <p>The screenshot shows the 'Word Ladder Solver' application window. The 'Start Word' is 'cat' and the 'Goal's Word' is 'dog'. The 'Choose Algorithm' dropdown is set to 'GBFS'. A 'Start Searching!' button is visible. Below it, the '4-Step Solution' is displayed: CAT, COT, DOT, and DOG. At the bottom, the status bar shows 'Path Length: 4', 'Visited Node's Amount: 4', and 'Execution Time: 0ms'.</p>



A\*

Word Ladder Solver
— □ ×

Start Word:

Goal's Word:

Choose Algorithm:

A\* ▼

Start Searching!

4-Step Solution

**CAT**  
**COT**  
**DOT**  
**DOG**

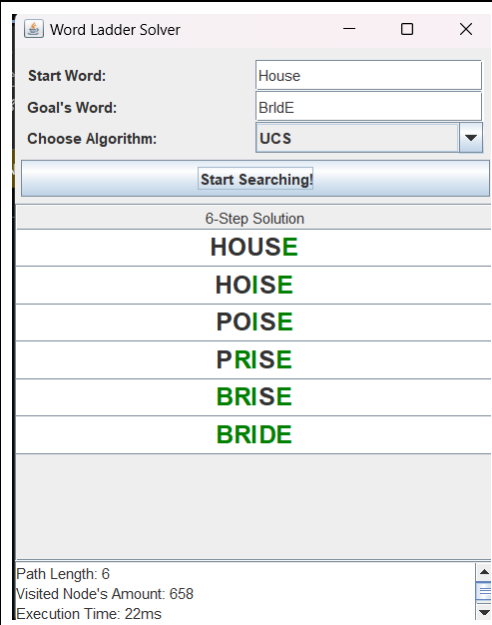
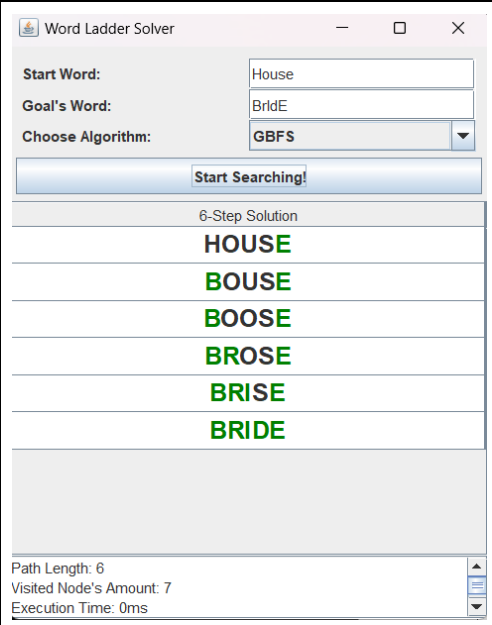
Path Length: 4  
 Visited Node's Amount: 6  
 Execution Time: 0ms

	Path Length	Visited Node's Amount	Execution time
UCS	4	624	12ms
GBFS	4	4	0<t<1 ms
A*	4	6	0<t<1 ms

Input ke-2

Kata Awal	House
Kata Akhir	BrIdE

Output

UCS	 <p>The screenshot shows the 'Word Ladder Solver' application window. The 'Start Word' is 'House', the 'Goal's Word' is 'BrIdE', and the 'Choose Algorithm' is set to 'UCS'. A 'Start Searching!' button is visible. Below it, a '6-Step Solution' is displayed in a list: HOUSE, HOISE, POISE, PRISE, BRISE, and BRIDE. At the bottom, statistics are shown: Path Length: 6, Visited Node's Amount: 658, and Execution Time: 22ms.</p>
GBFS	 <p>The screenshot shows the 'Word Ladder Solver' application window. The 'Start Word' is 'House', the 'Goal's Word' is 'BrIdE', and the 'Choose Algorithm' is set to 'GBFS'. A 'Start Searching!' button is visible. Below it, a '6-Step Solution' is displayed in a list: HOUSE, BOUSE, BOOSE, BROSE, BRISE, and BRIDE. At the bottom, statistics are shown: Path Length: 6, Visited Node's Amount: 7, and Execution Time: 0ms.</p>

A\*

Word Ladder Solver
— □ ×

Start Word:

Goal's Word:

Choose Algorithm:

A\*
▼

Start Searching!

6-Step Solution

HOUSE

HOISE

POISE

PRISE

BRISE

BRIDE

Path Length: 6  
 Visited Node's Amount: 16  
 Execution Time: 0ms

	Path Length	Visited Node's Amount	Execution time
UCS	6	658	22ms
GBFS	6	7	0<t<1 ms
A*	6	16	0<t<1 ms

### Input ke-3 (Edge Case)

Panjang kata awal  $\neq$  Panjang kata akhir

Kata Awal	Cat
Kata Akhir	kitten

Kata awal atau kata akhir tidak terisi

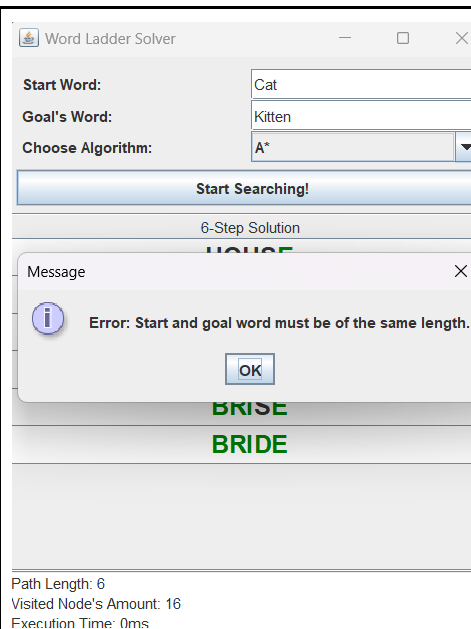
Kata Awal	
Kata Akhir	kitten

Kata tidak ada dalam kamus

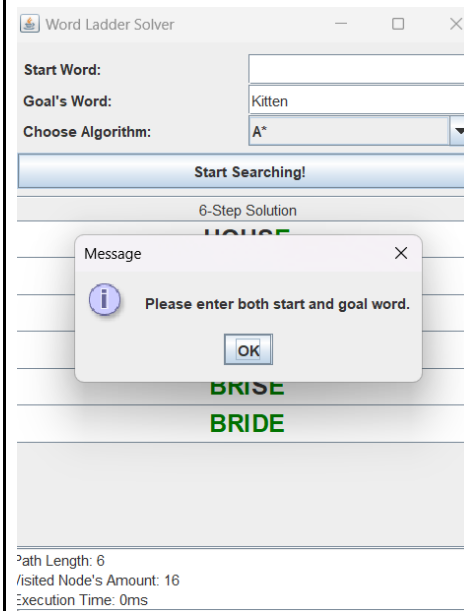
Kata Awal	Hehehe
Kata Akhir	Hahaha

### Output

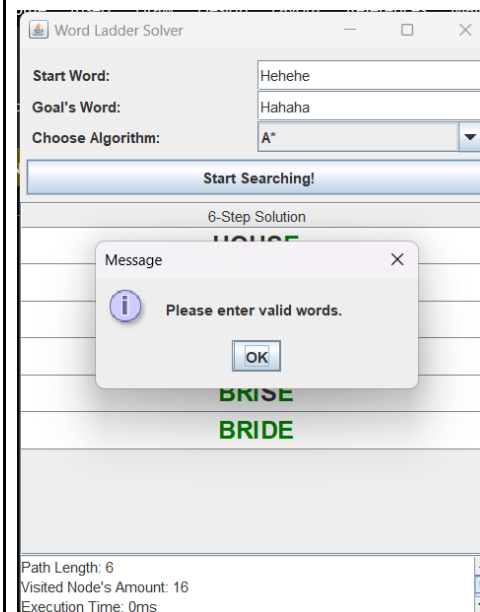
#### Error 1



Error 2



Error 3

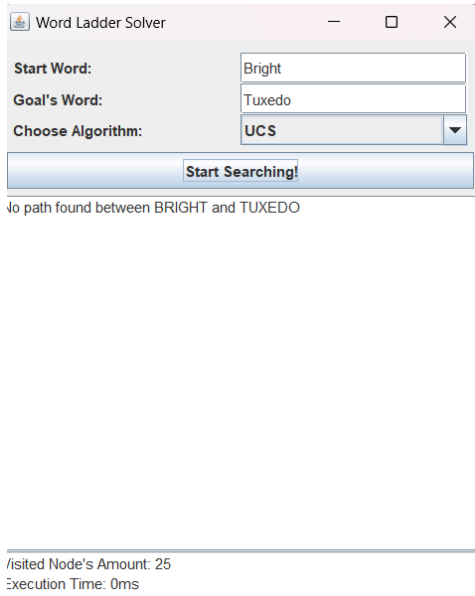
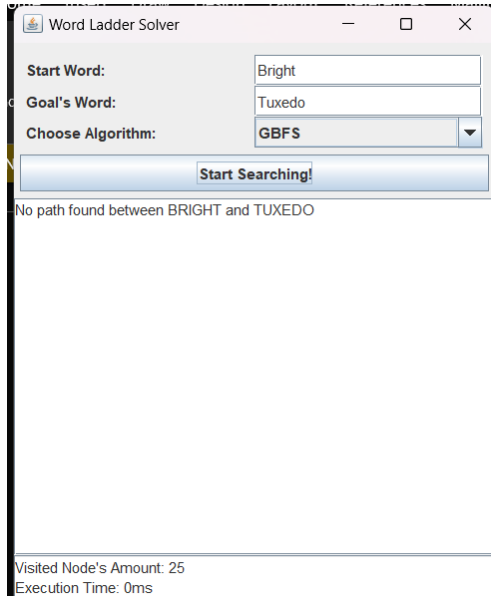


Input ke-4

Jika tidak ada path antara kata awal dan kata akhir

Kata Awal	Bright
Kata Akhir	Tuxedo

Output

UCS	
GBFS	

A\*

Word Ladder Solver

Start Word: Bright

Goal's Word: Tuxedo

Choose Algorithm: A\*

Start Searching!

No path found between BRIGHT and TUXEDO

Visited Node's Amount: 25  
Execution Time: 0ms

	Path Length	Visited Node's Amount	Execution time
UCS	-	25	0<t<1 ms
GBFS	-	25	0<t<1 ms
A*	-	25	0<t<1 ms

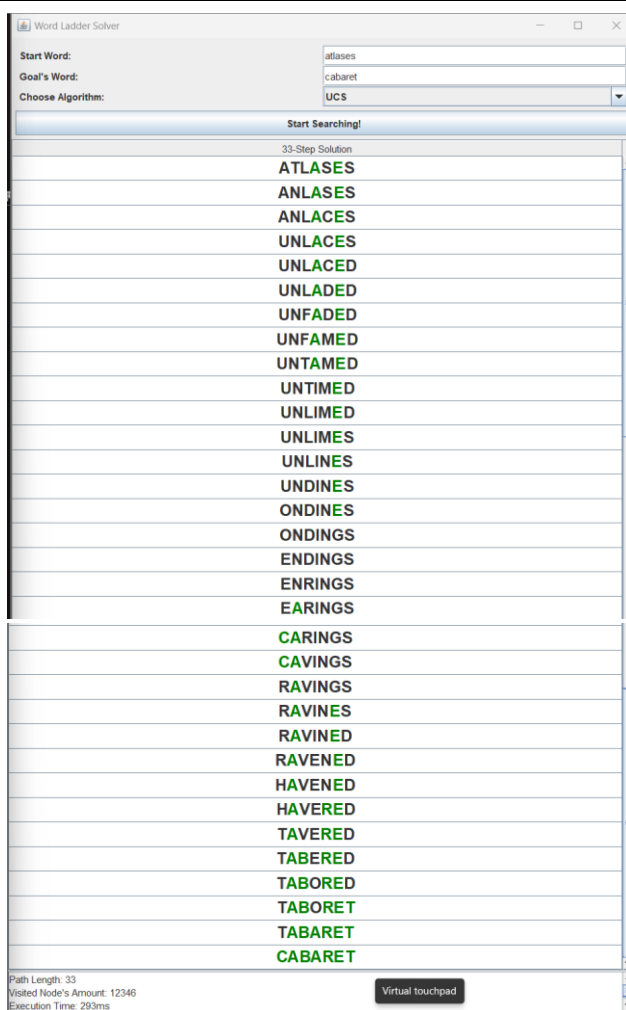
Input ke-5

Jika tidak ada path antara kata awal dan kata akhir

Kata Awal	Atlases
Kata Akhir	Cabaret

Output

UCS





## GBFS

The image displays two screenshots of the 'Word Ladder Solver' application, demonstrating the results of a GBFS (Goal-Breadth-First Search) algorithm. Both screenshots show a 66-step solution path from the start word 'atlases' to the goal word 'cabaret'.

**Top Screenshot:**

- Start Word:** atlases
- Goal's Word:** cabaret
- Choose Algorithm:** GBFS
- Start Searching!**
- 66-Step Solution:**
  - ATLASES
  - ANLASES
  - ANLACES
  - UNLACES
  - UNLACED
  - UNLAWED
  - UNSAWED
  - UNSAVED
  - UNPAVED
  - UNPARED
  - UNCARED
  - UNCAPED
  - UNCAPES
  - UNCASES
  - UNEASES
  - UREASES
  - CREASES
  - CREASED
  - CREAKED
  - CROAKED
- Path Length:** 66
- Visited Node's Amount:** 1495
- Execution Time:** 26ms

**Bottom Screenshot:**

- Start Word:** atlases
- Goal's Word:** cabaret
- Choose Algorithm:** GBFS
- Start Searching!**
- 66-Step Solution:**
  - CREAKED
  - CROAKED
  - CROCKED
  - CROCKET
  - CRICKET
  - CLICKET
  - CLICKER
  - CLINKER
  - CLINKED
  - CLINGED
  - CLANGED
  - CHANGED
  - CHANGER
  - CHANTER
  - CHUNTER
  - COUNTER
  - COURTER
  - COURTED
  - COURIED
  - COURIES
- Path Length:** 66

Word Ladder Solver

Start Word: atlases  
Goal's Word: cabaret  
Choose Algorithm: GBFS

Start Searching!

66-Step Solution

COORIES  
COOKIES  
COCKIES  
COCKLES  
CAKCKLES  
TACKLES  
TACKIES  
TALKIES  
WALKIES  
WALLIES  
BALLIES  
BALLSES  
BALISES  
VALISES  
VALINES  
SALINES  
SAVINES  
RAVINES  
RAVINED  
RAVENED  
HAVENED  
HAVERED  
TAVERED  
TABERED  
TABORED  
TABORET  
TABARET  
CABARET

Path Length: 66  
Visited Node's Amount: 1495  
Execution Time: 26ms

A\*

Word Ladder Solver

Start Word: atlases  
Goal's Word: cabaret  
Choose Algorithm: A\*

Start Searching!

33-Step Solution

ATLASES  
ANLASES  
ANLACES  
UNLACES  
UNLACED  
UNLADED  
UNFADED  
UNFAMED  
UNTAMED  
UNTIMED  
UNLIMED  
UNLIMES  
UNLINES  
UNDINES  
ONDINES  
ONDINGS  
ENDINGS  
ENRINGS  
EARINGS  
EATINGS

Path Length: 33  
Visited Node's Amount: 6456  
Execution Time: 36ms

EATINGS

RATINGS

RATINES

RAVINES

RAVINED

RAVENED

HAVENED

HAVERED

TAVERED

TABERED

TABORED

TABORET

TABARET

CABARET

Path Length: 33

Visited Node's Amount: 6456

Execution Time: 205ms

	Path Length	Visited Node's Amount	Execution time
UCS	33	12346	293ms
GBFS	66	1495	26ms
A*	33	6456	205ms

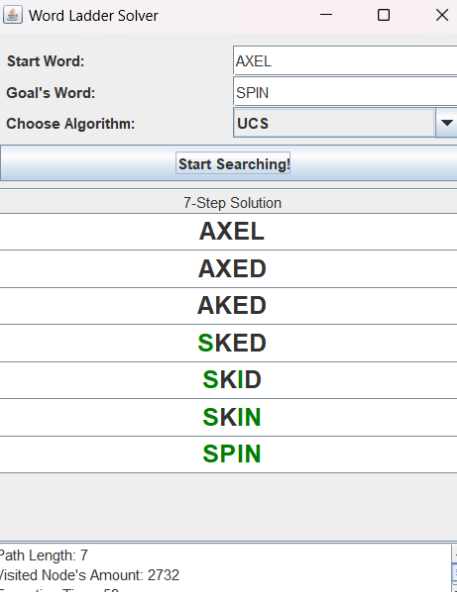
Input ke-6

Daily Word Ladder

Kata Awal	AXEL
Kata Akhir	SPIN

Output

UCS



Word Ladder Solver

Start Word: AXEL

Goal's Word: SPIN

Choose Algorithm: UCS

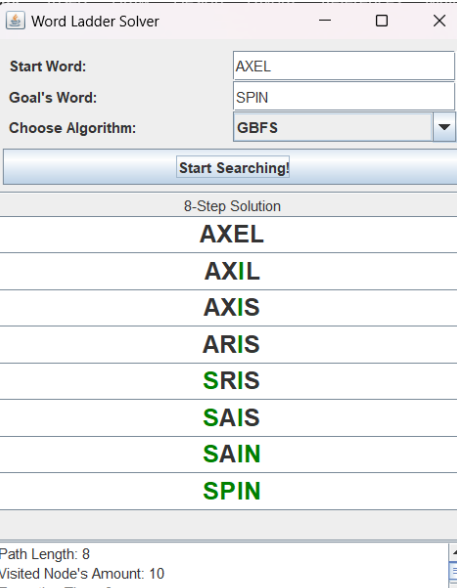
Start Searching!

7-Step Solution

AXEL
AXED
AKED
SKED
SKID
SKIN
SPIN

Path Length: 7  
Visited Node's Amount: 2732  
Execution Time: 58ms

GBFS



Word Ladder Solver

Start Word: AXEL

Goal's Word: SPIN

Choose Algorithm: GBFS

Start Searching!

8-Step Solution

AXEL
AXIL
AXIS
ARIS
SRIS
SAIS
SAIN
SPIN

Path Length: 8  
Visited Node's Amount: 10  
Execution Time: 0ms

A\*

The screenshot shows a window titled "Word Ladder Solver". It has three input fields: "Start Word:" with "AXEL", "Goal's Word:" with "SPIN", and "Choose Algorithm:" with a dropdown menu showing "A\*". Below these is a blue button labeled "Start Searching!". Underneath the button is a section titled "7-Step Solution" containing a list of words: AXEL, AXED, APED, SPED, SPUD, SPUN, and SPIN. The words from APED to SPIN are highlighted in green. At the bottom of the window, a status bar displays "Path Length: 7", "Visited Node's Amount: 61", and "Execution Time: 0ms".

	Path Length	Visited Node's Amount	Execution time
UCS	7	2732	58 ms
GBFS	8	10	0<t<1 ms
A*	7	61	0<t<1 ms

## 5. Analisis Perbandingan

### 5.1 Analisis Edge Case

Dari test case di atas, kondisi edge case sudah diperhitungkan, yakni

- Panjang start word dan goal word berbeda (Test Case 3 error 1)
- Start word dan/atau goal word tidak diisi (Test Case 3 error 2)
- Start word dan/atau goal word tidak berada di dalam kamus (Test Case 3 error 3)
- Start word dan goal word tidak membentuk jalur (Test Case 4)

### 5.2 Analisis panjang jalur

Dapat dilihat, untuk **kasus kecil** (Test Case 1, “Cat” ke “Dog”), **panjang jalur** yang dihasilkan oleh ketiga algoritma **sama**. Tetapi saat panjang kata semakin besar, maka **panjang jalur GBFS** cenderung **lebih panjang** dibanding UCS dan A\* yang memiliki panjang jalur yang **sama**. Contohnya ada pada Test Case 5 dan Test Case 6.

Sehingga dapat disimpulkan bahwa

- UCS dan A\* akan selalu menghasilkan solusi dengan panjang jalur yang optimal baik untuk kasus panjang kata kecil maupun panjang kata besar.
- GBFS akan menghasilkan solusi dengan panjang jalur yang optimal untuk kasus panjang kata kecil, namun cenderung tidak optimal untuk panjang kata besar.

### 5.3 Analisis Penggunaan Memori

Dapat dilihat, untuk kasus kecil sekalipun (Test Case 1), kebutuhan memori dalam bentuk banyak node yang dikunjungi oleh algoritma UCS selalu paling besar, lalu diikuti oleh algoritma A\* dan paling sedikit adalah GBFS. Banyak node yang dikunjungi algoritma A\* lumayan dekat dengan algoritma GBFS yang memiliki jarak cukup jauh dengan algoritma UCS. Hal ini dikarenakan UCS yang setara dengan algoritma BFS, yang menelusuri setiap node yang terbentuk di satu level sampai menemukan solusi, sementara A\* memiliki tambahan fungsi heuristik yang sangat membantu dalam mengefisiensikan penggunaan memori. Namun, GBFS lah yang paling sedikit dalam penggunaan memori karena hanya berfokus pada satu rute saja.

Sehingga dapat disimpulkan bahwa

- UCS adalah algoritma yang paling banyak menggunakan memori dalam pengeksekusiannya karena harus menelusuri setiap node yang diekspan.
- A\* dan GBFS memiliki fungsi heuristik yang sangat mengurangi jumlah node yang harus ditelusuri, dengan GBFS memiliki efisiensi lebih dibanding A\*

## 5.4 Analisis Waktu

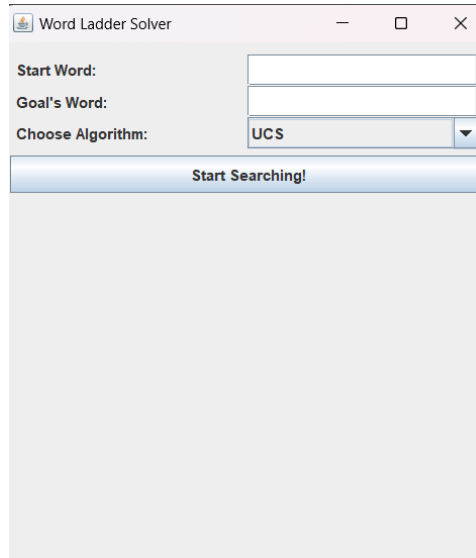
Waktu eksekusi program sangat bergantung dengan seberapa banyak hal yang harus diproses oleh komputer. Hal ini terefleksikan oleh seberapa banyak memori yang dibutuhkan untuk menyimpan hal yang harus diproses. Maka tidak heran bahwa algoritma UCS memiliki waktu yang lebih lambat jika dibandingkan dengan A\* dan GBFS, dan A\* masih lebih lambat dibandingkan GBFS tetapi tidak terpaut jauh.

Sehingga dapat disimpulkan bahwa

- UCS memiliki waktu eksekusi yang paling lambat dikarenakan jumlah hal yang harus diproses juga sangat banyak, mengingat penggunaan memorinya yang juga lebih banyak jika dibanding dengan algoritma lain
- A\* dan GBFS memiliki waktu eksekusi yang lebih cepat, dengan A\* sedikit lebih lambat dibanding GBFS karena jumlah hal yang harus diproses juga relatif lebih sedikit.

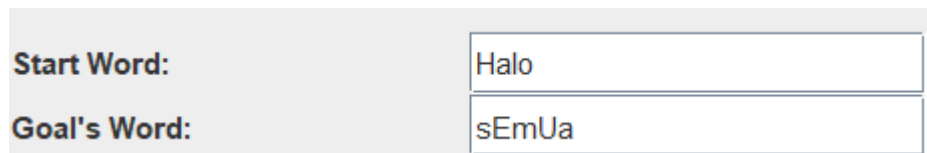
## 6. Implementasi Bonus

Implementasi bonus berupa GUI dibuat di dalam WordLadderGUI.java. Berikut adalah tampilan GUI yang telah dibuat.

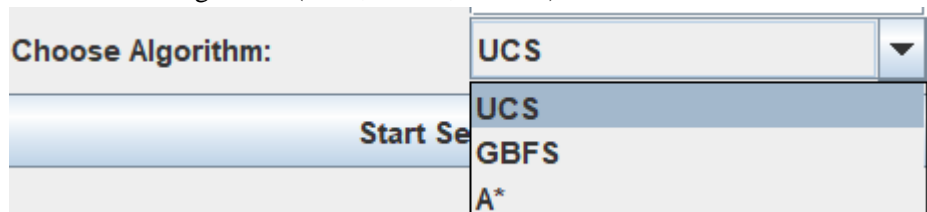


Beberapa fitur yang ada pada GUI:

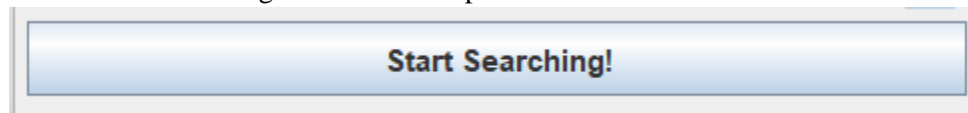
1. Start Word dan Goal Word



2. Pilihan Jenis Algoritma (UCS, GBFS, dan A\*)

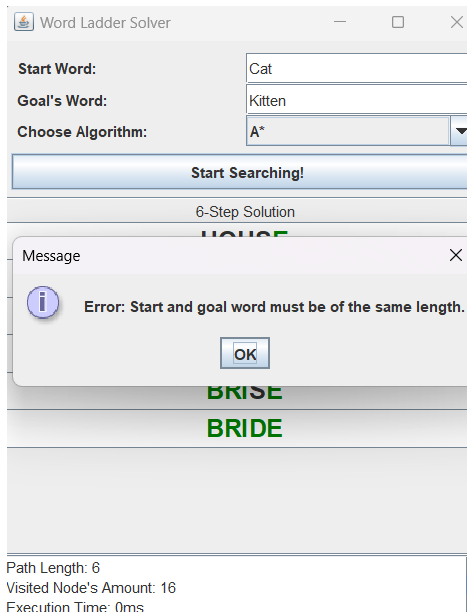


3. Tombol Start Searching untuk memulai pencarian Word Ladder

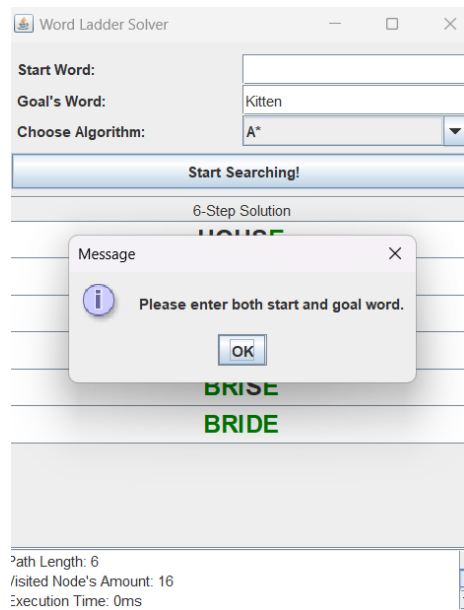


4. Jika masukan tidak valid, maka akan menampilkan pop-up message
  - a. Jika panjang kata tidak sama

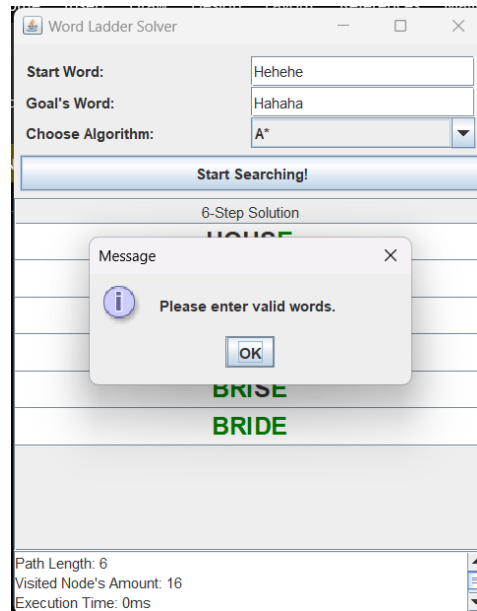




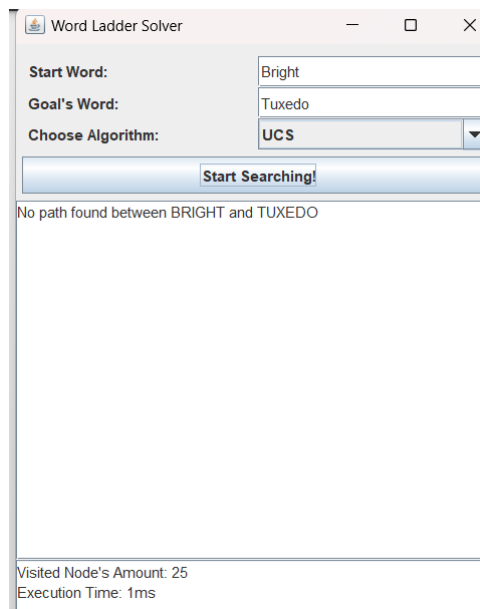
b. Jika Field Start word dan/atau Goal word tidak diisi



c. Jika kata tidak ditemukan di dalam kamus



5. Jika start word dan goal word tidak membentuk Word Ladder



6. Jika start word dan goal word membentuk Word Ladder, maka akan menampilkan solusi dan informasi terkait, berupa Path Length, Visited Node Ammount, dan Execution Time dalam ms

The screenshot shows a window titled "Word Ladder Solver". It contains input fields for "Start Word:" (Point), "Goal's Word:" (tight), and a dropdown for "Choose Algorithm:" (UCS). A "Start Searching!" button is below these fields. The results section, titled "11-Step Solution", lists the following words in a vertical list: POINT, PAINT, SAINT, STINT, STENT, SIENT, SIENS, SIGNS, and SIGHS. The letters in the words are color-coded: green for letters that change between adjacent words and black for letters that remain the same. At the bottom, the following statistics are displayed: Path Length: 11, Visited Node's Amount: 10835, and Execution Time: 243ms.

Step	Word
1	POINT
2	PAINT
3	SAINT
4	STINT
5	STENT
6	SIENT
7	SIENS
8	SIGNS
9	SIGHS

Path Length: 11  
Visited Node's Amount: 10835  
Execution Time: 243ms

# Lampiran

Progress

•

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	✓	

Link Repository Github

[https://github.com/Kharris-Khisunica/TUCIL3\\_13522051.git](https://github.com/Kharris-Khisunica/TUCIL3_13522051.git)