LAPORAN TUGAS KECIL III

IF2211 STRATEGI ALGORITMA

PENYELESAIAN PERMAINAN WORD LADDER MENGGUNAKAN ALGORITMA UCS, GREEDY BEST FIRST SEARCH, DAN A*

Disusun oleh:

William Glory Henderson 13522113

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

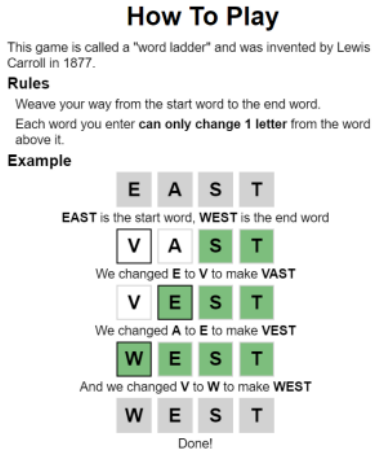
INSTITUT TEKNOLOGI BANDUNG

2024

Daftar Isi

Daftar Isi 2
BAB I DESKRIPSI MASALAH 3
BAB II LANDASAN TEORI 4
2.1 Algoritma Uniform Cost Search (UCS) 4
2.2 Algoritma Greedy Best First Search 4
2.3 Algoritma A* Search 5
BAB III IMPLEMENTASI PROGRAM 6
3.1 Main.java 6
3.2 WordLadderGUI.java 9
3.3 Dictionary.java 14
3.4 Graph.java 15
3.5 Node.java 16
3.6 WordSolver.java 17
BAB IV PENGUJIAN DAN ANALISIS 23
4.1 PENGUJIAN 23
4.1.1 Pengujian Test Case 1 23
4.1.2 Pengujian Test Case 2 26
4.1.3 Pengujian Test Case 3 28
4.1.4 Pengujian Test Case 4 31
4.1.5 Pengujian Test Case 5 34
4.1.6 Pengujian Test Case 6 37
4.2 ANALISIS
4.2.1 Hasil Pengujian 39
4.2.2 Analisis 40
BAB V IMPLEMENTASI BONUS 48
5.1 Penjelasan Bonus 48
BAB VI PENUTUP 49
6.1. Kesimpulan 49
6.2. Saran 50
DAFTAR PUSTAKA 51
LAMPIRAN 52

BAB I
DESKRIPSI MASALAH



Gambar 1. Ilustrasi dan Peraturan Permainan Word Ladder
(Sumber: https://wordwormdormdork.com/)

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. Pencarian solusi dari permainan Word Ladder bisa menggunakan beberapa algoritma pencarian seperti algoritma Uniform Cost Search (UCS), algoritma Greedy Best First Search, dan algoritma A* search.

BAB II
LANDASAN TEORI

2.1 Algoritma Uniform Cost Search (UCS)

Uniform Cost Search (UCS) merupakan jenis algoritma pencarian yang digunakan untuk menemukan jalur terpendek dengan fokus mencari biaya terendah dari titik awal ke titik tujuan. Algoritma ini tidak menggunakan heuristik, tetapi memperluas node dengan total biaya terendah yang diketahui dari node awal. Dalam UCS, $f(n) = g(n)$ yang mencerminkan biaya yang sudah dikeluarkan dari simpul awal untuk mencapai simpul *n*. UCS menggunakan struktur data *priority queue* untuk menyimpan semua simpul yang menunggu untuk ditelusuri dengan urutan simpul yang memiliki biaya kumulatif terendah diambil terlebih dahulu untuk diekspansi.

Setiap simpul yang diekspansi di UCS diperiksa untuk menentukan apakah itu merupakan simpul tujuan. Jika bukan, algoritma kemudian memperluas simpul tersebut dengan mengunjungi semua tetangganya yang belum dieksplorasi atau yang biaya melaluinya bisa diminimalisasi oleh rute yang baru ditemukan. Biaya untuk mencapai tetangga ini dihitung dengan menambahkan biaya dari simpul saat ini ke tetangga tersebut. Jika biaya baru ini lebih rendah dibandingkan dengan biaya yang sebelumnya tercatat untuk tetangga, biaya yang lebih baru akan menggantikan yang lama dan tetangga akan dimasukkan kembali ke dalam antrian dengan prioritas baru. Ini menjamin bahwa jalur yang ditemukan di setiap simpul adalah jalur dengan biaya terendah yang mungkin. Dalam konteks Word Ladder, UCS akan menjelajahi semua kemungkinan transformasi kata dengan biaya yang seragam (misalnya, setiap perubahan huruf dianggap memiliki biaya yang sama) dan mencari solusi dengan jumlah langkah terkecil.

2.2 Algoritma Greedy Best First Search

Greedy Best-First Search (GBFS) merupakan jenis algoritma pencarian yang mementingkan sebuah syarat sesuai yang diinginkan atau ditentukan dan yang terpenting adalah mencapai tujuan. Dalam GBFS, $f(n) = h(n)$ atau heuristic yang merupakan estimasi biaya dari node n ke tujuan. Heuristic yang dipakai pada kasus *word ladder*

solver adalah *hamming distance.* Solusi *Greedy Best-First Search* tidak akan selalu menjamin hasil yang optimal karena dapat terjebak dalam solusi optimum lokal, terutama dalam graf yang kompleks. Algoritma ini menggunakan heuristik untuk mengarahkan pencarian ke tujuan dengan cara yang paling efisien menurut perkiraan algoritma. Fokus utamanya adalah pencarian yang cepat ke tujuan tanpa memperhitungkan total biaya jalur dari awal. Algoritma ini selalu memilih untuk memperluas simpul yang tampaknya paling dekat dengan tujuan berdasarkan estimasi heuristik dari node saat ini ke tujuan tanpa mempertimbangkan biaya total dari simpul awal ke simpul saat ini. Sehingga algoritma ini merupakan algoritma yang paling cepat dibandingkan *A* Search* dan *UCS* tetapi solusinya belum tentu solusi yang optimal.

2.3 Algoritma A* Search

*A Search** merupakan algoritma pencarian yang memadukan algoritma UCS dan GBFS dalam mencari jalur terpendek. Algoritma A* menggunakan fungsi gabungan dari UCS dan GBFS yaitu $f(n) = g(n) + h(n)$ dimana $g(n)$ merupakan biaya awal ke node n dan $h(n)$ merupakan estimasi biaya dari node n ke tujuan. Fungsi $f(n)$ menggabungkan kedua nilai ini untuk setiap simpul dan simpul dengan nilai $f(n)$yang terkecil akan selalu dipilih untuk diekspansi berikutnya.

Heuristik untuk simpul berikutnya harus *admissible*, yaitu tidak pernah *overestimate* biaya sebenarnya atau tidak pernah *underestimate* biaya estimasi untuk mencapai tujuan. Ini menjamin bahwa solusi yang ditemukan tidak hanya efisien tetapi juga optimal. Heuristik yang digunakan dalam A* untuk *kasus word ladder solver* adalah *hamming distance*. A* ini menggabungkan keunggulan UCS dalam menemukan jalur dengan biaya minimal dan kecepatan GBFS dalam mendekati tujuan.

BAB III
IMPLEMENTASI PROGRAM

Berikut merupakan dari implementasi source code program, beserta penjelasan tiap class dan method yang diimplementasi.

3.1 Main.java

Main.java

```
import java.util.List;
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Dictionary dict = null;
        WordSolver solver = null;

        try {
            dict = new Dictionary("words.txt");
            solver = new WordSolver(dict);
        } catch (Exception e) {
            System.out.println("Failed to load dictionary");
        }

        System.out.println("Welcome to Word Ladder Solver!");
        System.out.println("Available algorithms: ");
        System.out.println("1. UCS (Uniform Cost Search)");
        System.out.println("2. GBFS (Greedy Best First Search)");
        System.out.println("3. Astar (A*)");
        System.out.println();
        System.out.print("Semua input akan otomatis diubah menjadi huruf kecil");

        String start, end;
        Scanner scanner = new Scanner(System.in);

        do {
            System.out.print("\nMasukkan kata awal: ");
            start = scanner.nextLine().toLowerCase();
            if (!dict.dictionary.contains(start)) {
```

```
            System.out.println("Kata '" + start + "' tidak tersedia di dalam kamus.");
        }
    } while (!dict.dictionary.contains(start));

    do {
        System.out.print("Masukkan kata akhir: ");
        end = scanner.nextLine().toLowerCase();
        if (!dict.dictionary.contains(end)) {
            System.out.println("Kata '" + end + "' tidak tersedia di dalam kamus.");
        } else if (start.length() ≠ end.length()) {
            System.out.println("Panjang kata tidak sama.");
        }
    } while (!dict.dictionary.contains(end) || start.length() ≠ end.length());

    int choice;
    do {
        System.out.print("\nPilih algoritma (1. UCS, 2. GBFS, 3. Astar): ");
        try {
            choice = Integer.parseInt(scanner.nextLine());
            if (choice < 1 || choice > 3) {
                throw new NumberFormatException();
            }
        } catch (NumberFormatException e) {
            System.out.println("Masukkan salah, Silahkan ulangi masukkan");
            choice = 0;
        }
    } while (choice < 1 || choice > 3);

    String algorithm;
    List<String> path = null;
    long startTime = System.currentTimeMillis();
    switch (choice) {
        case 1:
            algorithm = "ucs";
            path = solver.solve(start, end, algorithm, dict);
        case 2:
            algorithm = "gbfs";
            path = solver.solve(start, end, algorithm, dict);
        case 3:
            algorithm = "a*";
```

```
                path = solver.solve(start, end, algorithm, dict);
        }

        long endTime = System.currentTimeMillis();

        if (solver.getVisitedNodesCount() == 0) {
            System.out.println("No Solution");
        } else {
            System.out.println("Hasil path: ");
            int size = path.size();
            for (int i = 0; i < size; i++) {
                String word = path.get(i);
                System.out.println(word);
            }
            System.out.println("\nSolusi ditemukan dalam " + (endTime - startTime) + "
milidetik.");
            System.out.println("Jumlah node solusi: " + size);
            System.out.println("Jumlah node yang dikunjungi: " +
solver.getVisitedNodesCount());
        }
        scanner.close();
    }
}
```

Pada file Main.java terdapat class Main. Class Main memiliki static method main yang akan memanggil menjalankan program di terminal.

3.2 WordLadderGUI.java

WordLadderGUI.java

```
import javax.swing.*;
import java.awt.*;
import java.util.List;

public class WordLadderGUI extends JFrame {
    private JTextField startWordField;
    private JTextField endWordField;
    private JComboBox<String> algorithmSelector;
    private JPanel resultPanel;
    private JButton solveButton;
    private Dictionary dictionary;
    private WordSolver solver;
    private JLabel statusLabel;

    public WordLadderGUI() {
        createView();
        setTitle("Word Ladder Solver");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(700, 800);
        setLocationRelativeTo(null);
        setResizable(false);
        pack();
        setVisible(true);

        try {
            dictionary = new Dictionary("words.txt");
            solver = new WordSolver(dictionary);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(this, "Failed to load dictionary: " + e.getMessage(),
"Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    private void createView() {
        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());
```

```
        getContentPane().add(panel);

        Font font = new Font("SansSerif", Font.BOLD, 14);
        Font fontinput = new Font("SansSerif", Font.PLAIN, 13);
        Font fontselect = new Font("SansSerif", Font.BOLD, 13);

        JPanel inputPanel = new JPanel();
        inputPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 10, 10));
        startWordField = new JTextField(10);
        startWordField.setFont(fontinput);
        endWordField = new JTextField(10);
        endWordField.setFont(fontinput);
        algorithmSelector = new JComboBox<>(new String[]{"ucs", "gbfs", "a*"});
        algorithmSelector.setFont(fontselect);
        solveButton = new JButton("Solve");
        solveButton.setFont(fontselect);

        JLabel startWordLabel = new JLabel("Start Word:");
        startWordLabel.setFont(font);
        JLabel endWordLabel = new JLabel("End Word:");
        endWordLabel.setFont(font);
        JLabel algorithmLabel = new JLabel("Algorithm:");
        algorithmLabel.setFont(font);

        inputPanel.add(startWordLabel);
        inputPanel.add(startWordField);
        inputPanel.add(endWordLabel);
        inputPanel.add(endWordField);
        inputPanel.add(algorithmLabel);
        inputPanel.add(algorithmSelector);
        inputPanel.add(solveButton);
        panel.add(inputPanel, BorderLayout.NORTH);

        resultPanel = new JPanel();
        resultPanel.setLayout(new BoxLayout(resultPanel, BoxLayout.Y_AXIS));
        JScrollPane scrollPane = new JScrollPane(resultPanel);
        scrollPane.setPreferredSize(new Dimension(700, 700));
        scrollPane.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        panel.add(scrollPane, BorderLayout.CENTER);
```

```
        statusLabel = new JLabel(" ");
        statusLabel.setHorizontalAlignment(JLabel.CENTER);
        panel.add(statusLabel, BorderLayout.SOUTH);

        solveButton.addActionListener(e -> {
            solveButton.setEnabled(false);
            solve();
            solveButton.setEnabled(true);
        });
    }

    private void solve() {
        String startWord = startWordField.getText().trim().toLowerCase();
        String endWord = endWordField.getText().trim().toLowerCase();
        String algorithm = (String) algorithmSelector.getSelectedItem();
        if (startWord.isEmpty() || endWord.isEmpty()) {
            JOptionPane.showMessageDialog(this, "Please enter both words.", "Error",
JOptionPane.ERROR_MESSAGE);
            return;
        } else if (startWord.length() != endWord.length()) {
            JOptionPane.showMessageDialog(this, "Both size are not same", "Error",
JOptionPane.ERROR_MESSAGE);
            return;
        }

        solveButton.setEnabled(false);
        Runtime runtime = Runtime.getRuntime();
        long usedMemoryBefore = runtime.totalMemory() - runtime.freeMemory();
        long startTime = System.currentTimeMillis();

        try {
            List<String> path = solver.solve(startWord, endWord, algorithm, dictionary);
            long endTime = System.currentTimeMillis();
            long usedMemoryAfter = runtime.totalMemory() - runtime.freeMemory();
            long memoryUsed = Math.max(0, (usedMemoryAfter - usedMemoryBefore) / 1024);
            long duration = (endTime - startTime);

            if (path.isEmpty()) {
                memoryUsed = 0;
                duration = 0;
```

```
            }

            displayPath(path);
            statusLabel.setFont(new Font("SansSerif", Font.BOLD, 14));
            statusLabel.setText("<html><div style='padding-top: 10px; padding-bottom: 10px;
text-align: center;'>Solved in: " + duration + " ms   |   Nodes visited: "
+ solver.getVisitedNodesCount() +
                                "   |   Memory used: " + memoryUsed + "
</div></html>");
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(this, "Error solving word ladder: " +
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
        } finally {
            solveButton.setEnabled(true);
        }
    }

    private void displayPath(List<String> path) {
        resultPanel.removeAll();
        if (path.isEmpty()) {
            resultPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
            JLabel noSolutionLabel = new JLabel("No solution found");
            noSolutionLabel.setFont(new Font("Serif", Font.BOLD, 24));
            resultPanel.add(noSolutionLabel);
        } else {
            resultPanel.setLayout(new BoxLayout(resultPanel, BoxLayout.Y_AXIS));
            String prevWord = null;
            int size = path.size();
            for (int j = 0; j < size; j++) {
                String word = path.get(j);
                JPanel wordPanel = new JPanel(new GridLayout(1, word.length() + 1, 2, 0));
                wordPanel.setBorder(BorderFactory.createEmptyBorder(2, 2, 2, 2));
                JLabel numberLabel = new JLabel(String.valueOf(j + 1) + ".",
SwingConstants.CENTER);
                numberLabel.setPreferredSize(new Dimension(30, 30));
                wordPanel.add(numberLabel);

                for (int i = 0; i < word.length(); i++) {
                    JLabel letterLabel = new JLabel(String.valueOf(word.charAt(i)),
SwingConstants.CENTER);
```

```
                    letterLabel.setBorder(BorderFactory.createLineBorder(Color.BLACK));
                    letterLabel.setPreferredSize(new Dimension(30, 30));
                    letterLabel.setFont(letterLabel.getFont().deriveFont(18f));
                    if (prevWord ≠ null && word.charAt(i) ≠ prevWord.charAt(i)) {
                        letterLabel.setBackground(Color.GREEN);
                        letterLabel.setOpaque(true);
                    }
                    wordPanel.add(letterLabel);
                }
                if (j == 0) { // start word
                    wordPanel.setBackground(Color.LIGHT_GRAY);
                    JLabel startlabel = new JLabel("Start");
                    startlabel.setFont(new Font("Serif", Font.BOLD, 20));
                    resultPanel.add(startlabel);
                } else if (j == size - 1) { // stop word
                    resultPanel.add(Box.createVerticalStrut(20));
                    wordPanel.setBackground(Color.LIGHT_GRAY);
                    JLabel endlabel = new JLabel("End");
                    endlabel.setFont(new Font("Serif", Font.BOLD, 20));
                    resultPanel.add(endlabel);
                } else if (j == 1){
                    resultPanel.add(Box.createVerticalStrut(20));
                }
                resultPanel.add(wordPanel);
                prevWord = word;
            }
        }
        resultPanel.revalidate();
        resultPanel.repaint();
    }

    public static void main (String [] args){
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new WordLadderGUI();
            }
        });
    }
}
```

Pada file WordLadderGUI.java terdapat class WordLadderGUI. Class WordLadderGUI memiliki konstruktor yang akan menyiapkan tampilan GUI dan pembacaan file kamus untuk batasan kata dalam permainan kata serta menginisiasi method createView. Method createView ini akan meyiapkan tempat input start dan end word, tempat pemilihan algoritma, dan tombol untuk solver. Terdapat juga method solve yang akan menerima input kata serta proses error handling dalam input. Lalu juga akan ada penyimpanan run time dan memory yang dipakai. Terdapat method displayPath yang akan menampilkan perubahan kata dari awal hingga akhir. Terdapat juga static method main untuk menjalankan program dalam tampilan GUI.

3.3 Dictionary.java

Dictionary.java

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

public class Dictionary {
    public Set<String> dictionary;

    public Dictionary(String dictionaryFile) throws IOException {
        loadDictionary(dictionaryFile);
    }

    private void loadDictionary(String file) throws IOException {
        dictionary = new HashSet<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
            String line;
            while ((line = reader.readLine()) != null) {
                dictionary.add(line.trim().toLowerCase());
            }
        }
    }

    public boolean isValid(String word) {
```

```
        return dictionary.contains(word.toLowerCase());
    }
}
```

Pada file Dictionary.java terdapat class Dictionary. Class Dictionary memiliki method Dictionary untuk menginisasi kamus yang diberikan. Terdapat method loadDictionary yang digunakan untuk membaca file txt dan hasilnya disimpan dalam semua HashSet. Terdapat method isValid untuk memastikan bahwa kata yang disimpan dalam huruf kecil semua.

3.4 Graph.java

Graph.java

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

public class Graph {
    private Dictionary dictionary;

    public Graph(Dictionary dictionary) {
        this.dictionary = dictionary;
    }

    public List<String> getNeighbors(String word) {
        List<String> neighbors = new ArrayList<>();
        char[] chars = word.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            char oldChar = chars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                if (c != oldChar) {
                    chars[i] = c;
                    String newWord = new String(chars);
                    if (dictionary.isValid(newWord)) {
                        neighbors.add(newWord);
                    }
                }
            }
            chars[i] = oldChar;
```

```
        }
        return neighbors;
    }
}
```

Pada file Graph.java terdapat class Graph. Class Graph memiliki konstruktor untuk menginisialisasi Dictionary. Terdapat method getNeighbors untuk mencari tetangga dari sebuah kata. Maksud dari tetangga ini adalah kata perubahan yang mungkin dari suatu kata ke kata lain (cuma dapat berganti 1 huruf) dan dipastikan bahwa kata tersebut terdapat di dalam kamus.

3.5 Node.java

Node.java

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

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

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

Pada file Node.java terdapat class Node. Class Node berfungsi untuk menyimpan kata, parent dari kata tersebut, cost (biaya), dan heuristic. Atribut ini yang nantinya akan digunakan untuk perbandingan dalam mencari solusi serta untuk menampilkan hasil yang didapatkan.

3.6 WordSolver.java

WordSolver.java

```
import java.util.*;

public class WordSolver {
    private Graph graph;
    private int visitedNodesCount;

    public WordSolver(Dictionary dictionary) {
        this.graph = new Graph(dictionary);
    }

    // Choose the algorithm to solve
    public List<String> solve(String start, String end, String algorithm, Dictionary dict) {
        visitedNodesCount = 0;
        if (!dict.dictionary.contains(start) || !dict.dictionary.contains(end)) {
            throw new IllegalArgumentException("Start or end word not in dictionary.");
        }
        switch (algorithm.toLowerCase()) {
            case "ucs":
                return uniformCostSearch(start, end);
            case "gbfs":
                return greedyBestFirstSearch(start, end);
            case "a*":
                return aStarSearch(start, end);
            default:
                throw new IllegalArgumentException("Unknown algorithm type.");
        }
    }

    // UCS Algorithm
    public List<String> uniformCostSearch(String start, String end) {
        PriorityQueue<Node> prioqueue = new PriorityQueue<>(Comparator.comparingInt(n -> 
n.cost));
        Map<String, Integer> visitedCost = new HashMap<>();
        Set<String> allVisited = new HashSet<>(); // Track all visited nodes
        prioqueue.add(new Node(start, null, 0));
        visitedCost.put(start, 0);
```

```
        while (!prioqueue.isEmpty()) {
            Node current = prioqueue.poll();
            if (!allVisited.contains(current.word)){
                allVisited.add(current.word);
            }

            if (current.word.equals(end)) {
                visitedNodesCount = allVisited.size();
                return buildPath(current);
            }

            for (String neighbor : graph.getNeighbors(current.word)) {
                int newCost = current.cost + 1;
                if (!visitedCost.containsKey(neighbor) || visitedCost.get(neighbor) > newCost)
{
                    visitedCost.put(neighbor, newCost);
                    prioqueue.add(new Node(neighbor, current, newCost));
                }
            }
        }

        return Collections.emptyList();
    }

    // GBFS Algorithm 1
    // public List<String> greedyBestFirstSearch(String start, String end) {
    //     PriorityQueue<Node> prioqueue = new PriorityQueue<>(Comparator.comparingInt(n →
n.heuristic));
    //     Set<String> visited = new HashSet<>();
    //     Set<String> allVisited = new HashSet<>(); // Track all visited nodes
    //     prioqueue.add(new Node(start, null, 0, heuristic(start, end)));

    //     while (!prioqueue.isEmpty()) {
    //         Node current = prioqueue.poll();
    //         if (!allVisited.contains(current.word)){
    //             allVisited.add(current.word);
    //         }

    //         if (current.word.equals(end)) {
```

```
    //             visitedNodesCount = allVisited.size();
    //             return buildPath(current);
    //         }

    //         if (!visited.contains(current.word)) {
    //             visited.add(current.word);
    //             List<Node> neighbors = graph.getNeighbors(current.word).stream()
    //                 .filter(neighbor -> !visited.contains(neighbor))
    //                 .map(neighbor -> new Node(neighbor, current, current.cost + 1,
heuristic(neighbor, end)))
    //                 .sorted(Comparator.comparingInt(n -> n.heuristic))
    //                 .collect(Collectors.toList());

    //             if (!neighbors.isEmpty()) {
    //                 prioqueue.add(neighbors.get(0));
    //             }
    //         }
    //     }

    //     return Collections.emptyList();
    // }

    // GBFS Algorithm 2
    public List<String> greedyBestFirstSearch(String start, String end) {
        PriorityQueue<Node> prioqueue = new PriorityQueue<>(Comparator.comparingInt(n ->
n.heuristic));
        Set<String> visited = new HashSet<>();
        Set<String> allVisited = new HashSet<>(); // Track all visited nodes
        prioqueue.add(new Node(start, null, 0, heuristic(start, end)));

        while (!prioqueue.isEmpty()) {
            Node current = prioqueue.poll();
            if (!allVisited.contains(current.word)){
                allVisited.add(current.word);
            }

            if (current.word.equals(end)) {
                visitedNodesCount = allVisited.size();
                return buildPath(current);
            }
```

```
            if (!visited.contains(current.word)) {
                visited.add(current.word);
                for (String neighbor : graph.getNeighbors(current.word)) {
                    if (!visited.contains(neighbor)) {
                        prioqueue.add(new Node(neighbor, current, current.cost + 1,
heuristic(neighbor, end)));
                    }
                }
            }
        }

        return Collections.emptyList();
    }

    // A* Algorithm
    public List<String> aStarSearch(String start, String end) {
        PriorityQueue<Node> prioqueue = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost
+ n.heuristic));
        Map<String, Integer> visitedCost = new HashMap<>();
        Set<String> allVisited = new HashSet<>(); // Track all visited nodes
        prioqueue.add(new Node(start, null, 0, heuristic(start, end)));
        visitedCost.put(start, 0);

        while (!prioqueue.isEmpty()) {
            Node current = prioqueue.poll();
            if (!allVisited.contains(current.word)){
                allVisited.add(current.word);
            }

            if (current.word.equals(end)) {
                visitedNodesCount = allVisited.size();
                return buildPath(current);
            }

            for (String neighbor : graph.getNeighbors(current.word)) {
                int newCost = current.cost + 1;
                if (!visitedCost.containsKey(neighbor) || visitedCost.get(neighbor) > newCost)
{
                    visitedCost.put(neighbor, newCost);
```

```
                    prioqueue.add(new Node(neighbor, current, newCost, heuristic(neighbor,
end)));
                }
            }
        }

        return Collections.emptyList();
    }

    // Heuristic based on Hamming Distance
    public int heuristic(String current, String end) {
        int distance = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) != end.charAt(i)) {
                distance++;
            }
        }
        return distance;
    }

    public int getVisitedNodesCount() {
        return visitedNodesCount;
    }

    // Build the answer path
    public List<String> buildPath(Node endNode) {
        LinkedList<String> path = new LinkedList<>();
        for (Node n = endNode; n != null; n = n.parent) {
            path.addFirst(n.word);
        }
        return path;
    }
}
```

Pada file WordSolver.java terdapat class WordSolver. Class WordSolver memiliki atribut graph yang isinya kata-kata dari kamus dan visitedNodesCount untuk menandakan kata-kata yang ditelusuri selama dalam proses pencarian. Terdapat method WordSolver untuk menginisiasi kamus. Terdapat method solve untuk memanggil algoritma pencarian yang diinginkan pengguna.

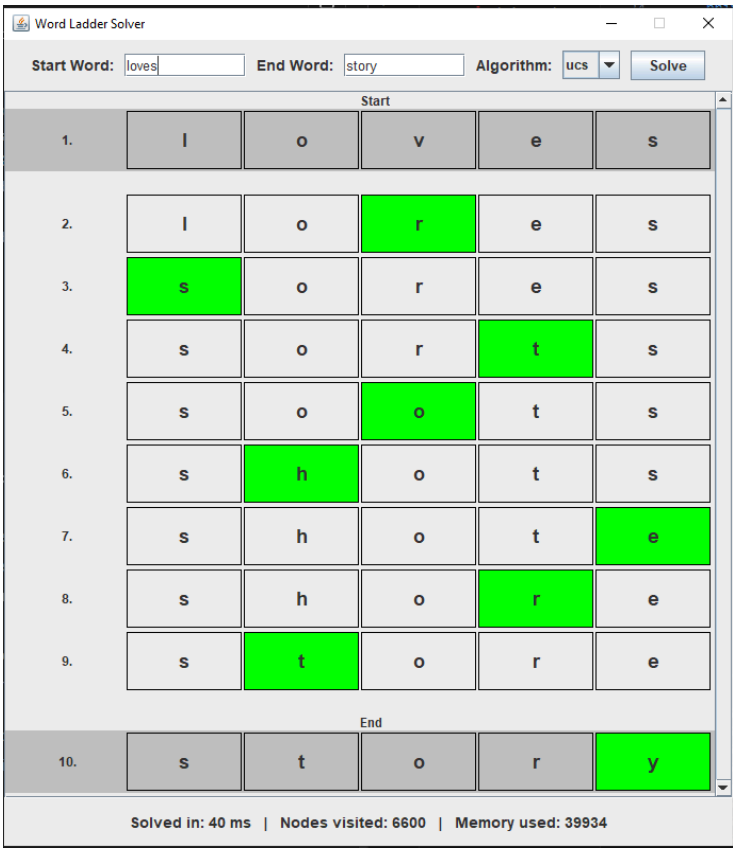
Terdapat method uniformCostSearch untuk melakukan pencarian dengan algoritma UCS. Terdapat method greedyBestFirstSearch untuk melakukan pencarian dengan algoritma GBFS. Terdapat method aStarSearch untuk melakukan pencarian dengan algoritma A*. Terdapat method heuristic untuk mencari nilai heuristic yang berdasarkan *hamming distance*. Terdapat juga method getVisitedNodes untuk mengembalikan jumlah kata yang ditelusuri dan terdapat method buildPath untuk mengembalikan path atau solusi jawaban yang didapatkan dari algoritma pencarian.

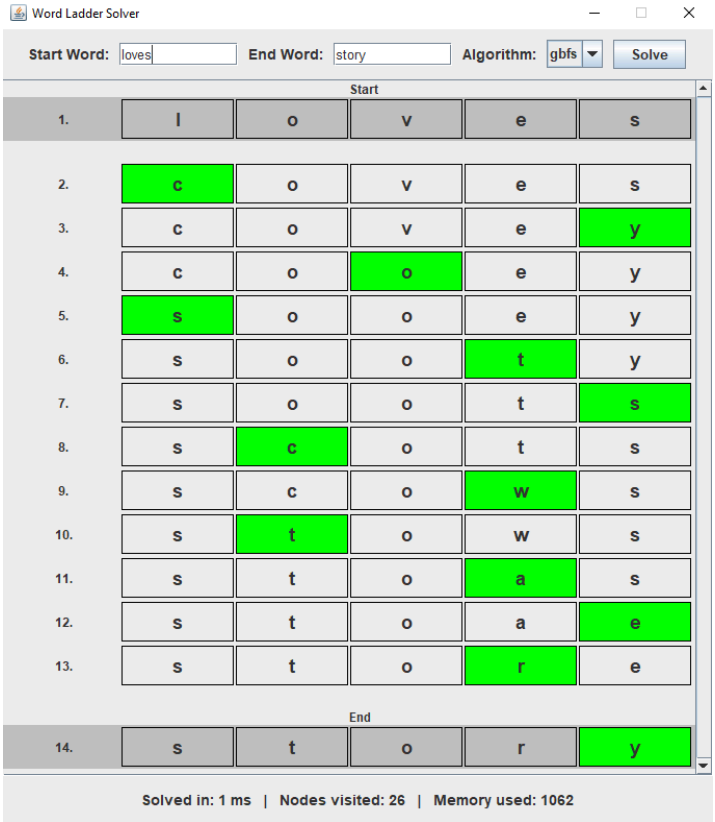
BAB IV
PENGUJIAN DAN ANALISIS

4.1 PENGUJIAN

4.1.1 Pengujian Test Case 1

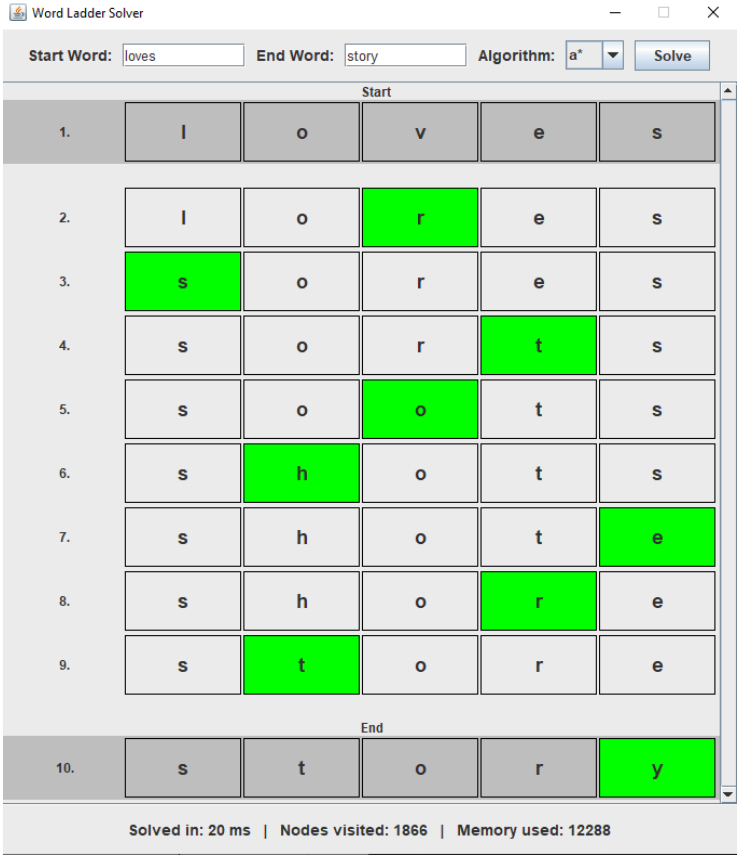
Tabel 1. Pengujian Test Case 1

Algoritma	Hasil Pengujian
UCS	

GBFS	

Word Ladder Solver
Start Word: loves
End Word: story
Algorithm: gbfs
Solve
Start
1. l o v e s
2. c o v e s
3. c o v e y
4. c o o e y
5. s o o e y
6. s o o t y
7. s o o t s
8. s c o t s
9. s c o w s
10. s t o w s
11. s t o a s
12. s t o a e
13. s t o r e
End
14. s t o r y
Solved in: 1 ms | Nodes visited: 26 | Memory used: 1062

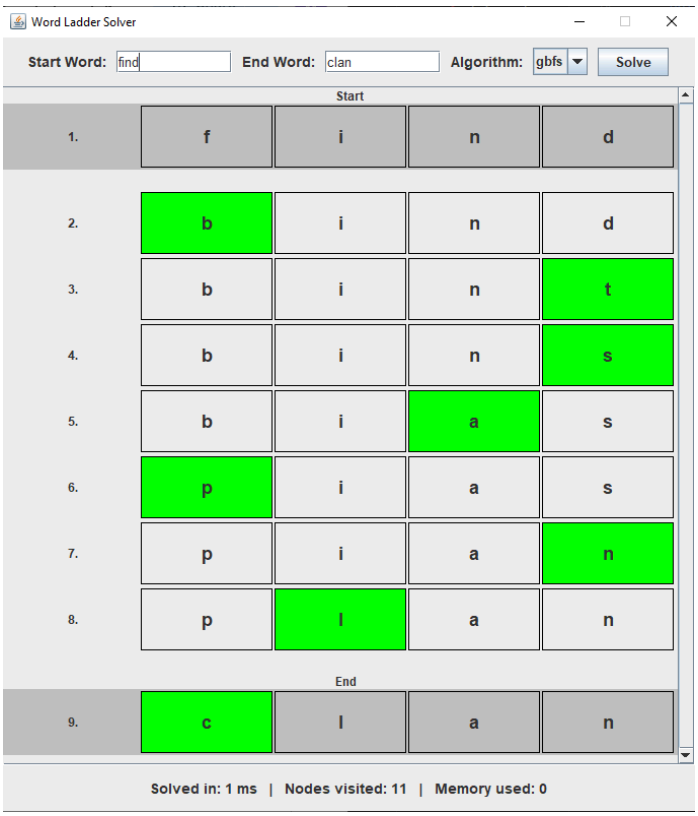
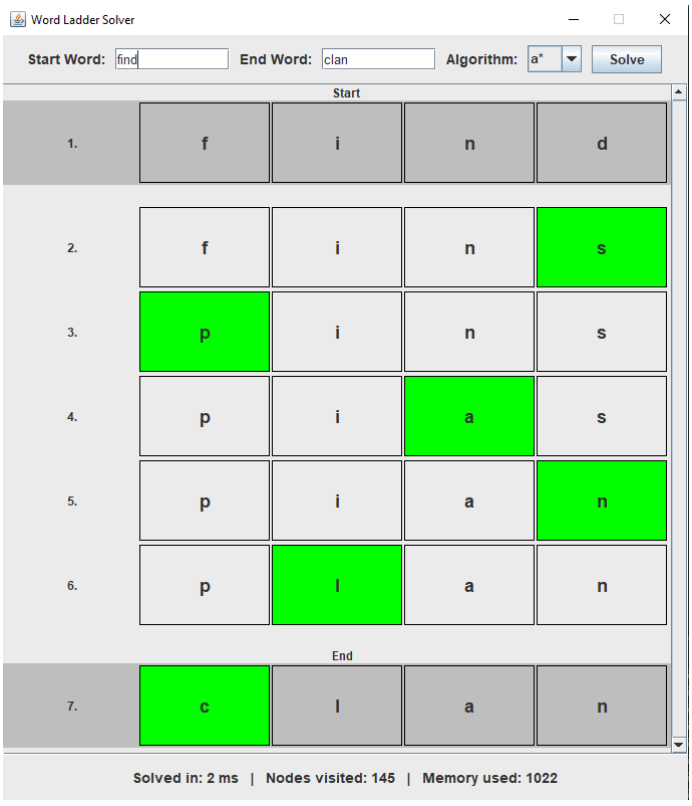
A*


Word Ladder Solver
Start Word: loves
End Word: story
Algorithm: a*
Solve
Start
1. l o v e s
2. l o r e s
3. s o r e s
4. s o r t s
5. s o o t s
6. s h o t s
7. s h o t e
8. s h o r e
9. s t o r e
End
10. s t o r y
Solved in: 20 ms | Nodes visited: 1866 | Memory used: 12288

4.1.2 Pengujian Test Case 2

Tabel 2. Pengujian Test Case 2

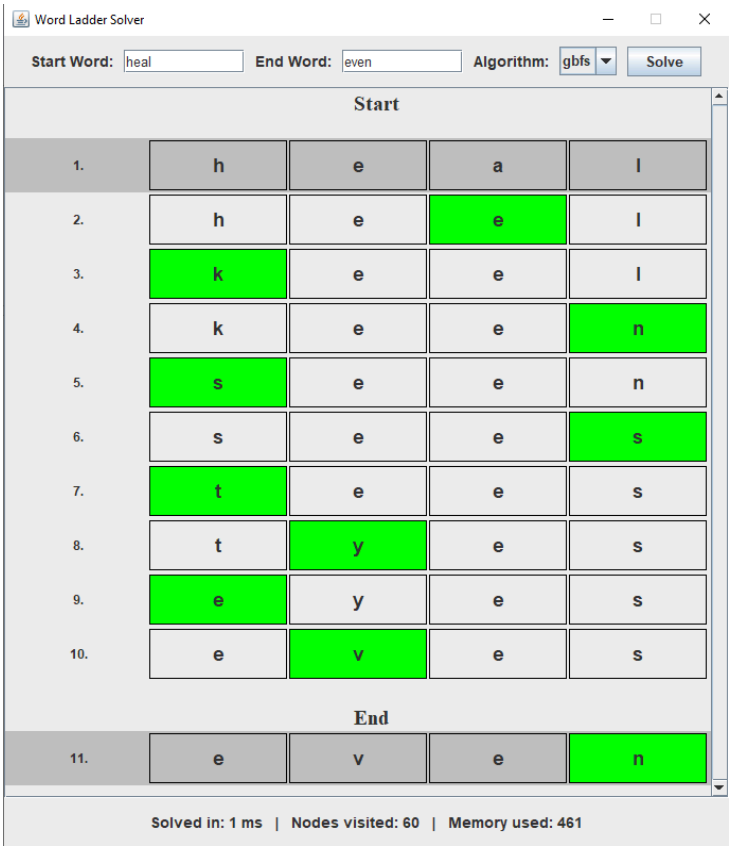
Algoritma	Hasil Pengujian
UCS	

GBFS	 Word Ladder Solver Start Word: find End Word: clan Algorithm: gbfs Solve Start 1. f i n d 2. b i n d 3. b i n t 4. b i n s 5. b i a s 6. p i a s 7. p i a n 8. p l a n End 9. c l a n Solved in: 1 ms | Nodes visited: 11 | Memory used: 0
A*	 Word Ladder Solver Start Word: find End Word: clan Algorithm: a* Solve Start 1. f i n d 2. f i n s 3. p i n s 4. p i a s 5. p i a n 6. p l a n End 7. c l a n Solved in: 2 ms | Nodes visited: 145 | Memory used: 1022

4.1.3 Pengujian Test Case 3

Tabel 3. Pengujian Test Case 3

Algoritma	Hasil Pengujian
UCS	

GBFS	

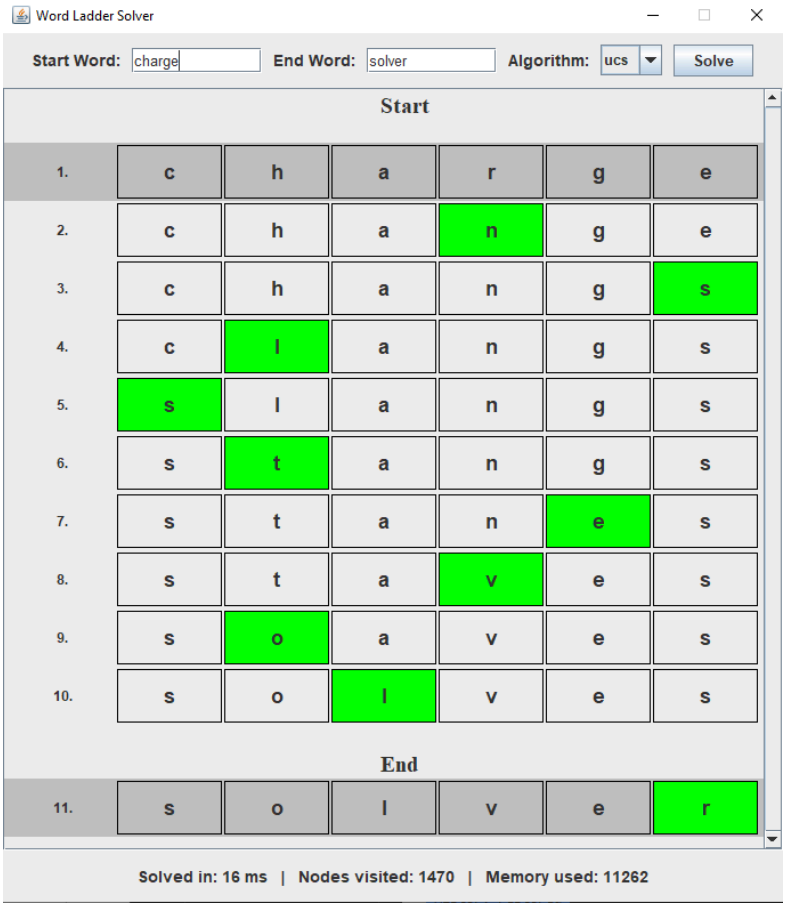
Word Ladder Solver
Start Word: heal
End Word: even
Algorithm: gbfs
Solve
Start
1. h e a l
2. h e e l
3. k e e l
4. k e e n
5. s e e n
6. s e e s
7. t e e s
8. t y e s
9. e y e s
10. e v e s
End
11. e v e n
Solved in: 1 ms | Nodes visited: 60 | Memory used: 461

A*	

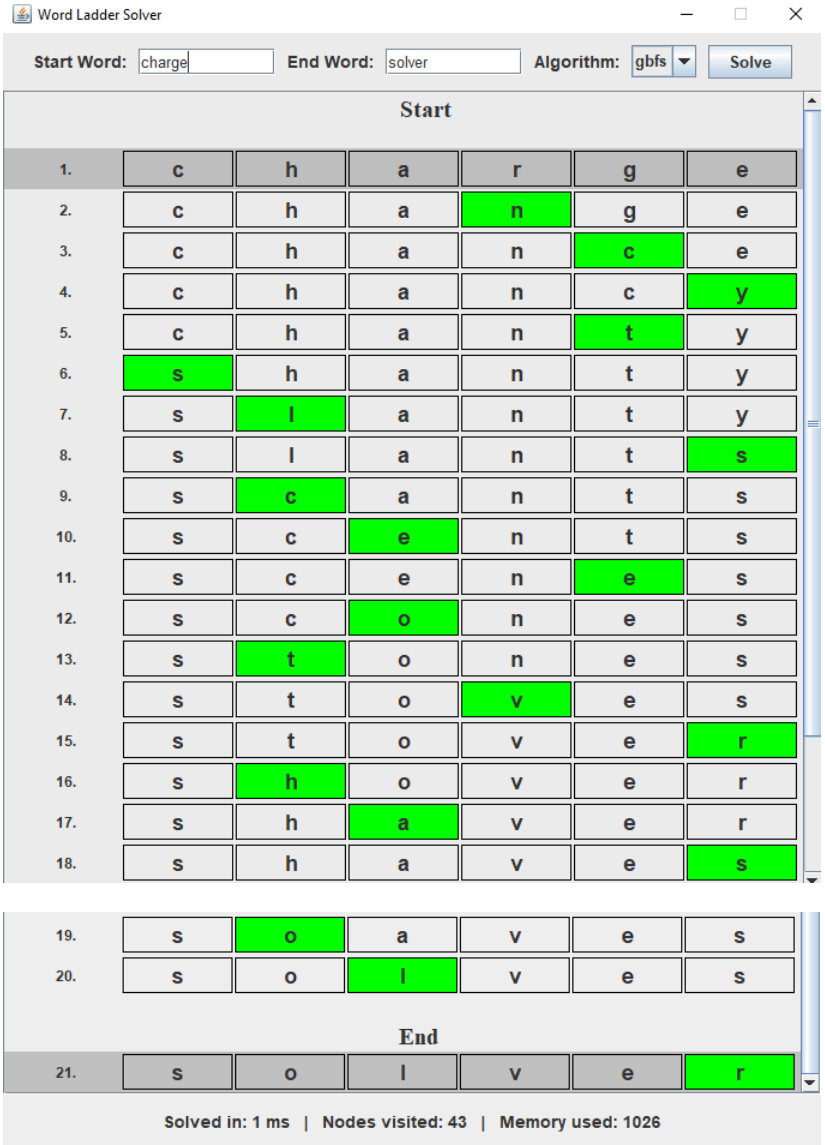
Word Ladder Solver
Start Word: heal
End Word: even
Algorithm: a*
Solve
Start
1. h e a l
2. h e e l
3. h e e d
4. d e e d
5. d y e d
6. e y e d
7. e y e n
End
8. e v e n
Solved in: 4 ms | Nodes visited: 421 | Memory used: 2048

4.1.4 Pengujian Test Case 4

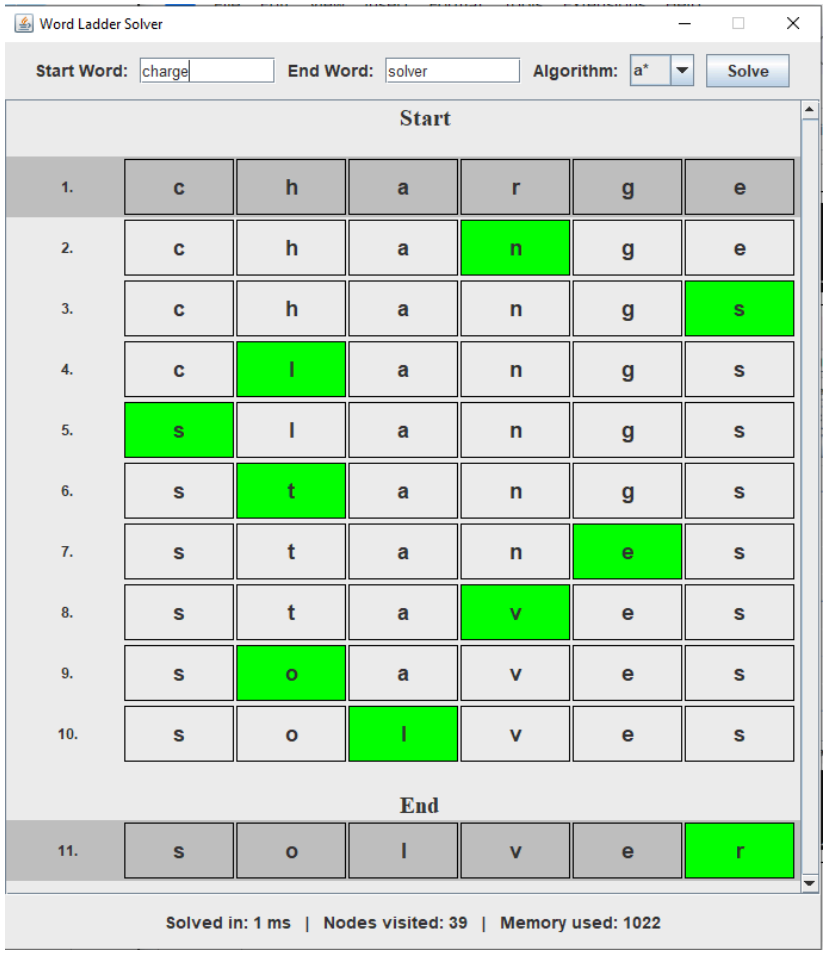
Tabel 4. Pengujian Test Case 4

Algoritma	Hasil Pengujian
UCS	

GBFS

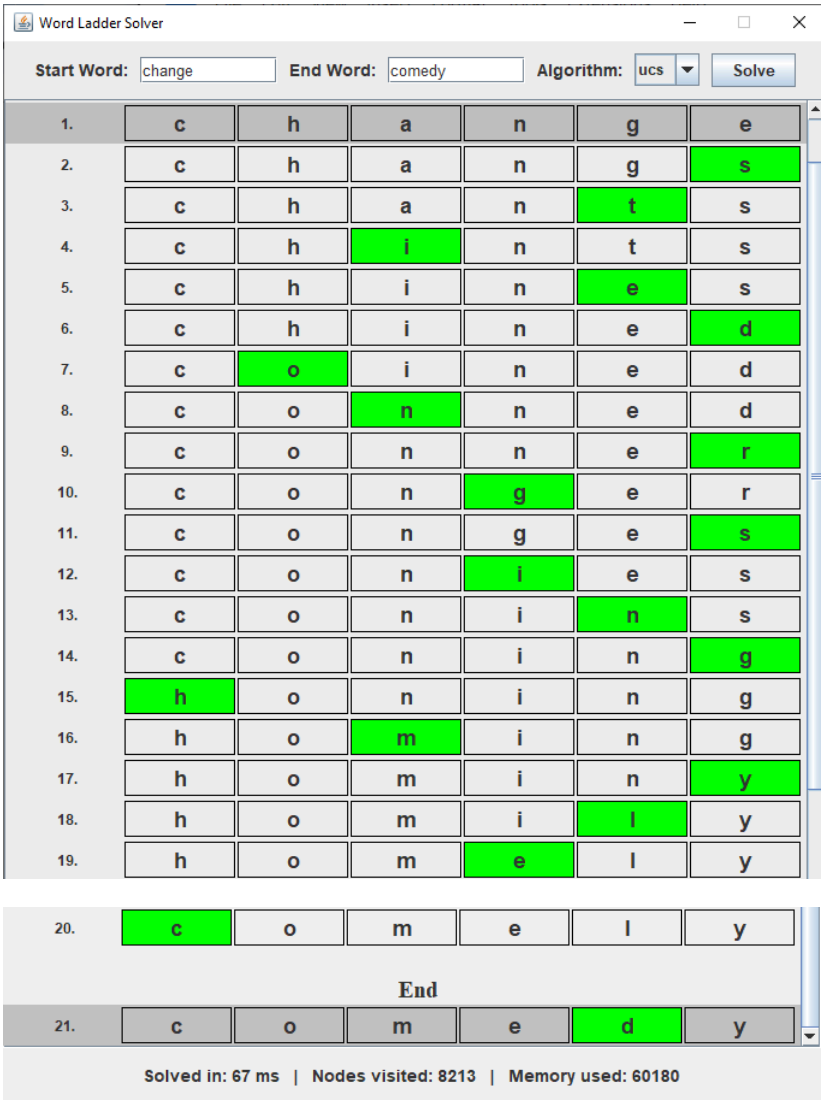

Word Ladder Solver
Start Word: charge
End Word: solver
Algorithm: gbfs
Solve
Start
1. c h a r g e
2. c h a n g e
3. c h a n c e
4. c h a n c y
5. c h a n t y
6. s h a n t y
7. s l a n t y
8. s l a n t s
9. s c a n t s
10. s c e n t s
11. s c e n e s
12. s c o n e s
13. s t o n e s
14. s t o v e s
15. s t o v e r
16. s h o v e r
17. s h a v e r
18. s h a v e s
19. s o a v e s
20. s o l v e s
End
21. s o l v e r
Solved in: 1 ms | Nodes visited: 43 | Memory used: 1026

A*

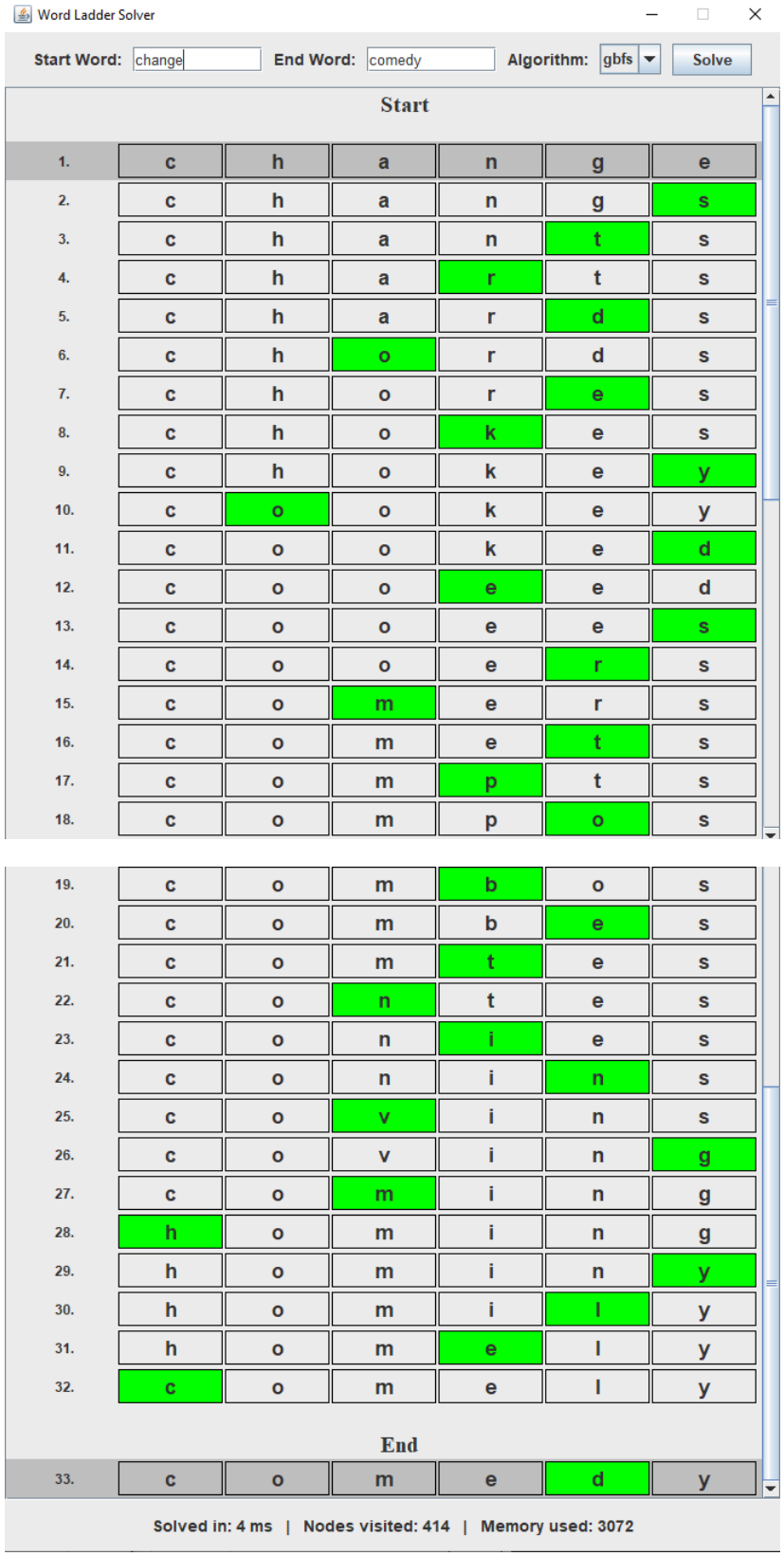

Word Ladder Solver
Start Word: charge
End Word: solver
Algorithm: a*
Solve
Start
1. c h a r g e
2. c h a n g e
3. c h a n g s
4. c l a n g s
5. s l a n g s
6. s t a n g s
7. s t a n e s
8. s t a v e s
9. s o a v e s
10. s o l v e s
End
11. s o l v e r
Solved in: 1 ms | Nodes visited: 39 | Memory used: 1022

4.1.5 Pengujian Test Case 5

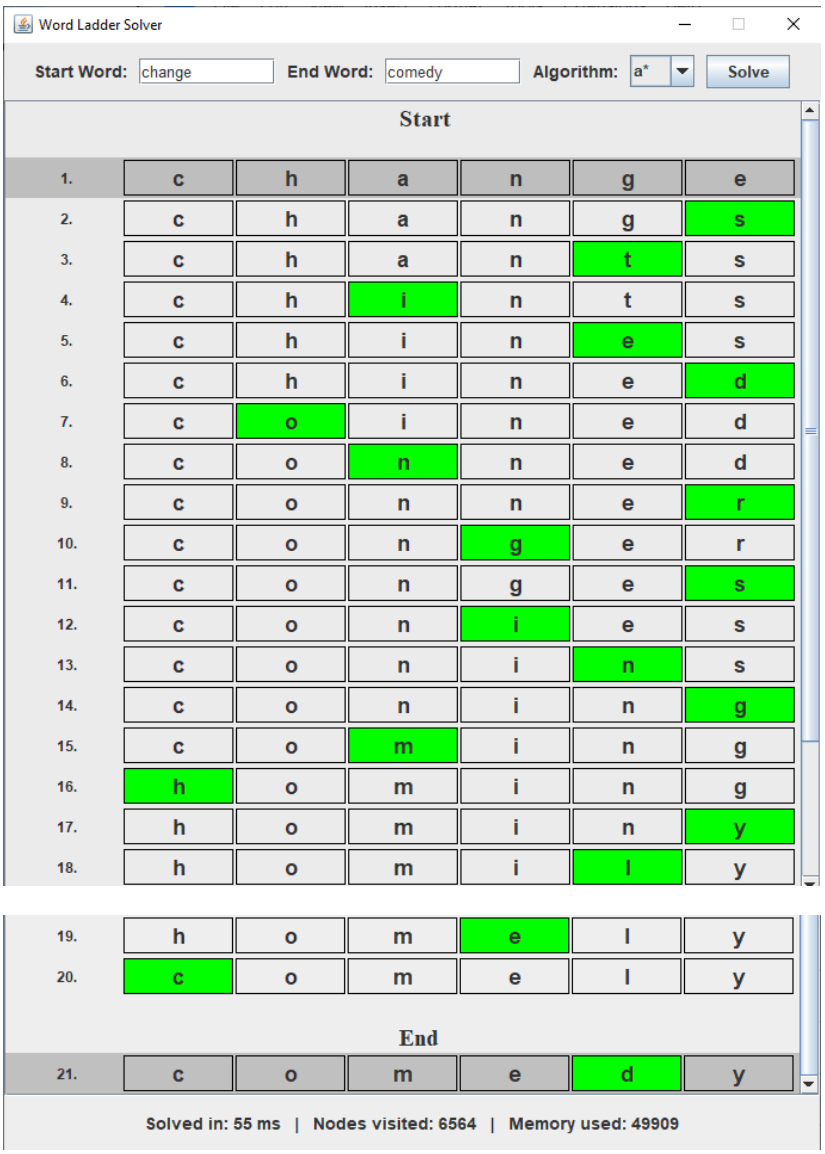
Tabel 5. Pengujian Test Case 5

Algoritma	Hasil Pengujian
UCS	

GBFS


Word Ladder Solver
Start Word: change
End Word: comedy
Algorithm: gbfs
Solve
Start
1. c h a n g e
2. c h a n g s
3. c h a n t s
4. c h a r t s
5. c h a r d s
6. c h o r d s
7. c h o r e s
8. c h o k e s
9. c h o k e y
10. c o o k e y
11. c o o k e d
12. c o o e e d
13. c o o e e s
14. c o o e r s
15. c o m e r s
16. c o m e t s
17. c o m p t s
18. c o m p o s
19. c o m b o s
20. c o m b e s
21. c o m t e s
22. c o n t e s
23. c o n i e s
24. c o n i n s
25. c o v i n s
26. c o v i n g
27. c o m i n g
28. h o m i n g
29. h o m i n y
30. h o m i l y
31. h o m e l y
32. c o m e l y
End
33. c o m e d y
Solved in: 4 ms | Nodes visited: 414 | Memory used: 3072

A*

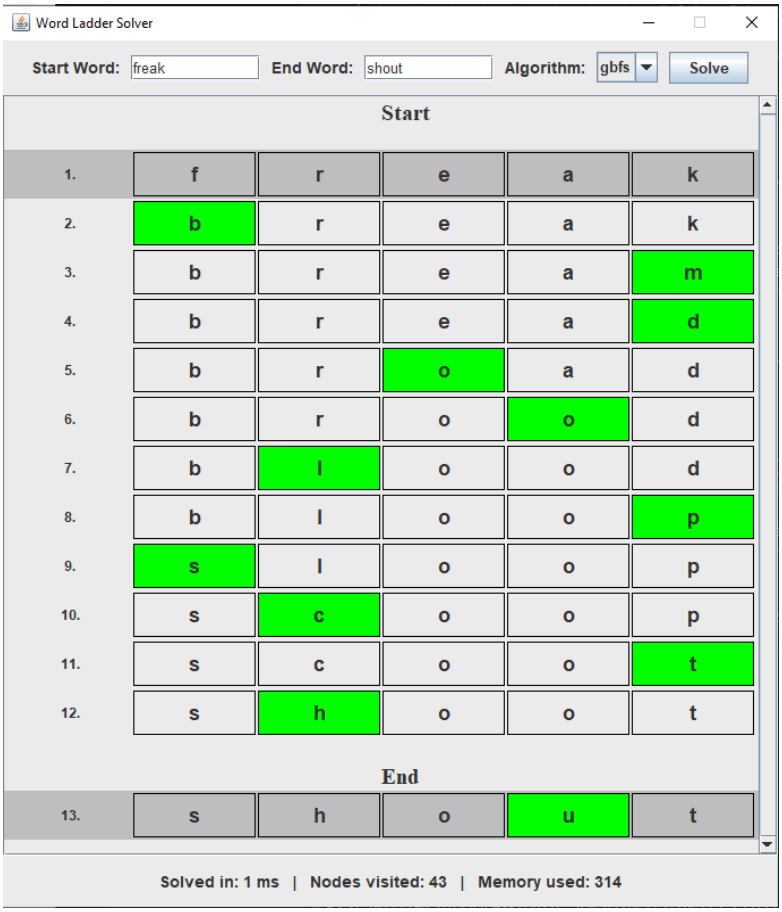

Word Ladder Solver
Start Word: change
End Word: comedy
Algorithm: a*
Solve
Start
1. c h a n g e
2. c h a n g s
3. c h a n t s
4. c h i n t s
5. c h i n e s
6. c h i n e d
7. c o i n e d
8. c o n n e d
9. c o n n e r
10. c o n g e r
11. c o n g e s
12. c o n i e s
13. c o n i n s
14. c o n i n g
15. c o m i n g
16. h o m i n g
17. h o m i n y
18. h o m i l y
19. h o m e l y
20. c o m e l y
End
21. c o m e d y
Solved in: 55 ms | Nodes visited: 6564 | Memory used: 49909

4.1.6 Pengujian Test Case 6

Tabel 6. Pengujian Test Case 6

Algoritma	Hasil Pengujian
UCS	

GBFS


Word Ladder Solver
Start Word: freak
End Word: shout
Algorithm: gbfs
Solve
Start
1. f r e a k
2. b r e a k
3. b r e a m
4. b r e a d
5. b r o a d
6. b r o o d
7. b l o o d
8. b l o o p
9. s l o o p
10. s c o o p
11. s c o o t
12. s h o o t
End
13. s h o u t
Solved in: 1 ms | Nodes visited: 43 | Memory used: 314

A*	

4.2 ANALISIS

4.2.1 Hasil Pengujian

Tabel 4.2.1 Hasil Pengujian Semua Test Case

Test Case	Algoritma	Jumlah Path Hasil	Jumlah Node yang Dikunjungi	Waktu Eksekusi	Jumlah Memory yang Dipakai
1	UCS	10	6600	40	39934
	GBFS	14	26	1	1062
	A*	10	1886	20	12288

2	UCS	7	3424	18	17410
	GBFS	9	11	1	0
	A*	7	145	2	1022
3	UCS	8	3531	25	17894
	GBFS	11	60	1	461
	A*	8	421	4	2048
4	UCS	11	1470	16	11262
	GBFS	21	43	1	1026
	A*	11	39	1	1022
5	UCS	21	8213	67	60180
	GBFS	33	414	4	3072
	A*	21	6564	55	49909
6	UCS	8	432	5	2596
	GBFS	13	43	1	314
	A*	8	23	0	308

4.2.2 Analisis

Langkah-Langkah Algoritma UCS untuk menyelesaikan permainan world ladder adalah sebagai berikut:

1. PriorityQueue prioqueue diinisialisasi untuk menyimpan node yang berisi kata, biaya, dan referensi ke node sebelumnya. Node disimpan berdasarkan biaya terendah (cost) yang dihitung dari jumlah penggantian huruf dari kata awal. Map

visitedCost diinisialisasi untuk menyimpan biaya terkecil untuk setiap kata yang telah dikunjungi. Set allVisited diinisialisasi untuk mencatat semua kata yang telah diproses.

2. Kata awal ditambahkan ke dalam prioqueue dan visitedCost dengan biaya 0.
3. Selama prioqueue tidak kosong, node dengan biaya terendah diambil dari antrian (disebut current).
4. Jika kata dalam node current belum terdapat dalam allVisited, tambahkan ke dalam allVisited.
5. Cek apakah kata current sama dengan kata tujuan (end). Jika ya, fungsi buildPath dipanggil untuk merekonstruksi jalur dari kata awal ke kata akhir dan mengembalikan hasilnya.
6. Untuk setiap kata tetangga yang valid (dihasilkan dengan mengganti satu huruf dari kata current), hitung biaya baru (newCost), yang merupakan biaya current ditambah satu.
7. Jika kata tetangga belum pernah dikunjungi atau biaya baru lebih rendah dari biaya yang tercatat di visitedCost, maka tambahkan visitedCost dengan newCost. Buat node baru untuk tetangga dengan biaya newCost dan tambahkan ke prioqueue . Ulangi langkah 3 sampai prioqueue kosong.
8. Jika prioqueue kosong dan kata tujuan tidak ditemukan, kembalikan daftar kosong, menandakan tidak ada jalur yang ditemukan.

Langkah-Langkah Algoritma GBFS untuk menyelesaikan permainan world ladder adalah sebagai berikut:

1. PriorityQueue prioqueue diinisialisasi untuk menyimpan node, dengan prioritas berdasarkan nilai heuristik dari setiap node. Nilai heuristik mewakili perkiraan "jarak" atau "biaya" dari node tersebut ke node tujuan. Set visited diinisialisasi untuk mencatat node yang sudah sepenuhnya dieksplorasi. Set allVisited diinisialisasi untuk melacak semua node yang telah dikunjungi, termasuk yang belum sepenuhnya dieksplorasi.
2. Kata awal ditambahkan ke dalam prioqueue dengan heuristik yang dihitung dari fungsi heuristic(start, end) dan biaya awal 0. Heuristic berdasarkan *hamming*

distance (jumlah langkah yang dibutuhkan untuk mengubah kata saat ini ke kata tujuan)

3. Selama prioqueue tidak kosong, node dengan heuristik terendah diambil dari antrian (disebut current).
4. Jika kata dalam node current belum terdapat dalam allVisited, tambahkan ke dalam allVisited.
5. Cek apakah kata current sama dengan kata tujuan (end). Jika ya, fungsi buildPath dipanggil untuk merekonstruksi jalur dari kata awal ke kata akhir dan mengembalikan hasilnya.
6. Jika kata dalam node current belum sepenuhnya dieksplorasi (visited belum mengandung kata current), tandai sebagai sudah dieksplorasi.
7. Untuk setiap tetangga yang valid dari kata current (dapat dihasilkan dengan mengganti satu huruf dari kata current dan belum terdapat dalam visited), maka tambahkan tetangga ke prioqueue dengan heuristik yang dihitung dari fungsi heuristic(neighbor, end) dan biaya current.cost + 1. Ulangi langkah 3 sampai prioqueue kosong.
8. Jika prioqueue kosong dan kata tujuan tidak ditemukan, kembalikan daftar kosong, menandakan tidak ada jalur yang ditemukan.

Langkah-Langkah Algoritma A* untuk menyelesaikan permainan world ladder adalah sebagai berikut:

1. PriorityQueue prioqueue diinisialisasi untuk menyimpan node Node diurutkan berdasarkan jumlah cost (biaya aktual dari start ke node saat ini) dan heuristic (estimasi biaya dari node saat ini ke end). Ini memastikan bahwa node dengan biaya total terendah (f(n) = g(n) + h(n)) selalu diproses terlebih dahulu. Map visitedCost diinisialisasi untuk memetakan setiap kata yang dikunjungi dengan biaya terendah untuk mencapainya. Ini digunakan untuk memeriksa apakah node yang sama dapat dicapai dengan biaya yang lebih rendah. Set allVisited diinisialisasi untuk menyimpan semua kata yang telah dikunjungi untuk tujuan pelaporan atau analisis.

2. Kata awal ditambahkan ke prioqueue dengan biaya nol dan heuristik yang dihitung menggunakan heuristic(start, end). Kata awal juga dimasukkan ke dalam visitedCost dengan biaya 0.
3. Selama PriorityQueue tidak kosong, node dengan total biaya (biaya sebenarnya dan heuristik) terendah diambil dari antrian (disebut current).
4. Jika kata dalam node current belum terdapat dalam allVisited, tambahkan ke dalam allVisited.
5. Cek apakah kata current sama dengan kata tujuan (end). Jika ya, fungsi buildPath dipanggil untuk merekonstruksi jalur dari kata awal ke kata akhir dan mengembalikan hasilnya.
6. Jika kata dalam node current belum sepenuhnya dieksplorasi (belum tercatat dalam visitedCost atau bisa dicapai dengan biaya yang lebih rendah dari yang telah dicatat sebelumnya)
7. Untuk setiap tetangga yang valid dari kata current (dapat dihasilkan dengan mengganti satu huruf dari kata current). Hitung biaya baru (newCost) yang merupakan current.cost + 1.
8. Jika tetangga belum terdapat dalam visitedCost atau newCost lebih rendah dari yang telah dicatat, perbarui visitedCost dan tambahkan node baru untuk tetangga tersebut ke prioqueue dengan newCost dan heuristik yang dihitung dari heuristic(neighbor, end). Ulangi langkah 3 sampai prioqueue kosong.
9. Jika prioqueue kosong dan kata tujuan tidak ditemukan, kembalikan daftar kosong, menandakan tidak ada jalur yang ditemukan.

Metode UCS memastikan bahwa solusi yang ditemukan adalah solusi optimal dalam hal jumlah perubahan huruf dari kata awal ke kata akhir karena selalu memprioritaskan simpul dengan cost terkecil untuk dieksplorasi terlebih dahulu. Dalam UCS, $f(n)$ adalah identik dengan $g(n)$ dimana $g(n)$ merupakan jumlah biaya yang dibutuhkan untuk mencapai simpul n atau simpul saat ini dari simpul awal. Pada program ini, simpul maksudnya sebagai kata. Pada algoritma ini, perubahan dari satu kata ke satu kata lain (perubahan sebuah huruf) dianggap biayanya sebesar 1. Perubahan kata ini dengan syarat kata yang baru tersebut terdapat dalam kamus. Simpul dengan cost terendah selalu diutamakan dan memastikan bahwa algoritma tidak

hanya mencapai tujuan tetapi melakukannya dengan jumlah langkah yang paling efisien. Meskipun perubahan satu kata ke kata lain dapat ditempuh dengan berbagai langkah, algoritma UCS akan mengambil langkah yang paling pendek untuk dijadikan solusi optimalnya.

Metode Greedy Best First Search menggunakan pendekatan heuristik berbasis hamming distance (berdasarkan perbedaan huruf). Dalam GBFS, $f(n)$ adalah identik dengan $h(n)$, di mana $h(n)$ adalah estimasi biaya dari simpul n ke tujuan. Pada konteks ini, $h(n)$ didapat dari menghitung banyaknya huruf yang tidak cocok saat membandingkan kata yang sedang dicek dengan kata tujuan. Algoritma ini memilih simpul yang dianggap paling dekat atau tampak mengarah langsung dengan tujuan untuk dieksplorasi berikutnya. Namun, Greedy Best First Search cenderung terjebak dalam minimum lokal dan tidak dapat memperbaiki keputusan yang sudah diambil. Sehingga solusi yang diberikan tidak optimal karena GBFS tidak mempertimbangkan keseluruhan jalur dan biaya dari awal.

Metode A* *Search* merupakan perpaduan antara UCS dan GBFS. Dalam A*, $f(n) = g(n) + h(n)$, yang memadukan biaya yang sudah dikeluarkan dari awal sampai saat ini $g(n)$ dengan estimasi biaya saat ini ke tujuan $h(n)$. Dalam konteks algoritma A*, $h(n)$ yang biasa disebut juga heuristik, dihitung berdasarkan perbedaan huruf yang tidak cocok dari simpul n ke tujuan. Sementara itu, $g(n)$ adalah biaya aktual yang telah dikeluarkan dari simpul awal hingga simpul n.. Dengan menggunakan nilai $f(n)$, $g(n)$, dan $h(n)$ algoritma A* dapat memprioritaskan simpul yang memiliki nilai $f(n)$ yang lebih rendah, yang mengindikasikan kemungkinan jalur yang lebih baik atau lebih efisien. Algoritma A* sangat dihargai karena keefektifannya dalam memprioritaskan eksplorasi node yang tidak hanya dekat dengan titik awal tetapi juga yang tampak mendekati tujuan.

Heuristik pada metode A* didasarkan pada jumlah huruf yang tidak tepat atau sama antara dua kata. Pada algoritma A*, heuristik dianggap admissible jika fungsi heuristik tidak pernah melebihi biaya yang sebenarnya (*underestimate* biaya sebenarnya*)* untuk mencapai tujuan terdekat dari suatu node. Admissible syaratnya biaya estimasi ke tujuan $\leq$ biaya sebenarnya. Biaya di sini merupakan langkah minimal yang diperlukan untuk mengubah satu kata menjadi kata lain dengan syarat kata tersebut valid dalam kamus. Contoh kasus, untuk kata BASE ke WEST, heuristicnya adalah 3 karena ada 3 huruf yang harus diubah. Jawabannya adalah perubahan huruf E ke T, W ke B, A ke E. Untuk kasus ini dianggap admissible karena langkah

(biaya sebenarnya) yang dibutuhkan tidak kurang dari batas estimasi biaya ke tujuan dan setiap perubahan adalah kata yang valid. Akan tetapi, terkadang perubahan huruf tersebut tidak selalu menghasilkan kata yang valid sehingga membutuhkan langkah-langkah lain yang dapat melebihi estimasi biaya awal. Sehingga heuristik pada algoritma ini dapat dianggap admissible karena terkadang membutuhkan lebih banyak langkah (perubahan huruf) untuk mencapai kata tujuan dengan syarat kata yang valid.

Pada kasus *word ladder*, algoritma UCS berperilaku mirip BFS karena pada kasus ini nilai perubahan setiap 1 huruf dianggap biayanya 1. Sehingga algoritma UCS akan menjelajahi semua kemungkinan perubahan huruf pada level yang sama sebelum berpindah ke level berikutnya. Contoh kata BASE ke WEST, membutuhkan 3 kali perubahan maka biayanya 3 akan tetapi urutan perubahannya tidak dipedulikan tetapi biayanya harus paling kecil. Karena biaya perubahan kata yang sama, maka cara kerja node yang dibangkitkan UCS sama dengan BFS yaitu membangun semua kemungkinan pada satu level dan menjelajahi semua kemungkinan pada level tersebut baru lanjut ke level berikutnya. Untuk hasil pathnya tidak selalu sama karena dalam proses pembangunannya mungkin terdapat perbedaan urutan pada satu level yang sama akan tetapi proses hasilnya menghasilkan biaya yang sama. Tetapi untuk konteks word solver, karena terdapat acuan satu kamus yang sama, maka hasil path UCS dan BFS akan sama.

Algoritma A* secara teori lebih efisien dibandingkan algoritma UCS karena A* menggabungkan biaya yang dikeluarkan dari awal sampai saat ini (cost) dan estimasi biaya saat ini ke kata tujuan (heuristic). Algoritma A* akan jauh lebih efisien jika heuristik yang digunakan juga efisien dan admissible. UCS tidak lebih efisien karena perilakunya mirip BFS yang dalam kasus ini menjelajahi semua node baru lanjut ke level berikutnya dimana hal tersebut mirip *brute-force* yang menjelajahi semua kemungkinan tanpa memikirkan kemungkinan mana yang mencapai tujuan paling cepat. Sedangkan A* menggunakan heuristik untuk memperkirakan biaya saat ini ke tujuan, sehingga algoritma ini dapat memprioritaskan node mana yang lebih mungkin untuk mencapai tujuan dengan cepat. A* juga lebih efisien karena tidak perlu menjelajahi semua node seperti UCS karena A* memfokuskan pencarian pada path yang terlihat lebih menjanjikan. Heuristik yang buruk dapat mengurangi kinerja A*. Karena heuristik yang admissible juga, yaitu biaya sebenarnya tidak akan kurang dari biaya estimasi ke tujuan, maka

A* dijamin akan menemukan solusi yang optimal. Karena penggabungan yang dimiliki A*, pencarian dapat dilakukan lebih cepat dan efisien dibandingkan UCS karena tidak perlu menjelajahi semua kemungkinan dan dapat memprioritaskan kata yang lebih dekat dengan tujuan.

Algoritma GBFS secara teori lebih cepat dibandingkan algoritma UCS dan A* karena langsung mengarah ke kata yang mirip dengan kata tujuan. Tetapi GBFS seringkali mengabaikan jalan yang ditempuh sehingga terkadang jalan yang dipilih membutuhkan lebih banyak langkah akan tetapi proses pencariannya tidak perlu menelusuri semua kemungkinan. Artinya, GBFS dapat bekerja lebih cepat daripada UCS dan A* tetapi hasil yang dihasilkan belum tentu solusi yang optimal. Algoritma GBFS seringkali terjebak dalam optimum lokal karena GBFS tidak mempertimbangkan biaya yang dikeluarkan dari awal dan hanya mementingkan estimasi biaya ke tujuan. Hal ini membuat GBFS melewatkan jalur yang secara keseluruhan biayanya lebih sedikit sehingga solusinya belum tentu solusi paling optimal.

Berdasarkan tabel hasil pengujian, Algoritma UCS dan A* selalu memberikan jumlah solusi path yang sama dan solusi tersebut adalah solusi optimal. Hal ini dikarenakan mempertimbangkan biaya awal sampai akhir yang paling sedikit. Algoritma GBFS tidak memberikan solusi path yang optimal tetapi ada kasus dimana GBFS dapat memberikan solusi path optimal ketika perbedaan huruf hanya 1 atau 2 kata awal ke kata tujuan dan perubahan tersebut adalah kata valid. Untuk jumlah path yang ditelusuri paling banyak adalah UCS karena hal ini perlu menjelajahi seluruh kemungkinan node pada suatu level lalu baru lanjut ke level selanjutnya. GBFS memiliki jumlah path yang ditelusuri paling sedikit karena GBFS hanya berfokus pada mencapai tujuan tanpa mempertimbangkan dari awal. Algoritma A* tidak selalu memiliki jumlah path yang ditelusuri paling sedikit karena A* berusaha mencari solusi optimal sehingga A* perlu melakukan eksplorasi node lebih jika biayanya belum paling sedikit. Untuk pemakaian memory ini berbanding lurus dengan jumlah node yang ditelusuri. Semakin banyaknya node yang ditelusuri maka memory yang digunakan akan semakin banyak. Hal ini juga sebanding dengan run time. Run time paling lama adalah UCS karena jumlah node yang ditelusuri paling banyak dibandingkan A* dan GBFS. Run time GBFS paling cepat karena GBFS hanya mementingkan biaya estimasi ke tujuan sehingga GBFS tidak mementingkan biaya yang

sudah dikeluarkan dari awal. Untuk run time A* lebih cepat dari UCS karena A* akan memprioritaskan node yang dijelajahi berdasarkan heuristik tetapi tetap mempertimbangkan biaya yang telah dikeluarkan. Rata-rata dari 6 test case yang bisa diambil adalah solusi paling optimal A* dan UCS. Untuk runtime paling cepat dan memory paling sedikit adalah GBFS. Untuk runtime paling lama dan memory paling besar adalah UCS. Sehingga jika membutuhkan solusi yang optimal tetapi dengan pemakaian memory yang sedikit dan runtime yang tidak lama, dapat dikatakan bahwa A* dapat bekerja lebih efisien dan solusinya optimal.

BAB V
IMPLEMENTASI BONUS

5.1 Penjelasan Bonus

Bonus yang diimplementasikan adalah bonus GUI atau tampilan antarmuka. GUI yang dibuat ini menggunakan javax swing yang merupakan library dari bahasa java. Pada Class WordLadderGUI ini terbagi menjadi 1 konstruktor untuk menginisialisasi GUI dan 3 method untuk menampilkan fitur-fitur. Ketika program dijalankan, akan muncul kotak atau canvas berukuran 700 x 800 (h x w). Bagian atas akan terdapat kotak input untuk start word dan end word, terdapat kotak untuk pemilihan algoritma dan terdapat dropdown suggestion, dan terdapat tombol solve untuk memulai pencarian. Bagian atas ini terdapat dalam method createView. Kemudian ada method solve, method yang berguna untuk menampilkan error handling ketika input tidak ada dalam kamus, kata yang diinput hanya 1 atau tidak ada, dan size antara kedua kata berbeda. Lalu juga menampilkan berapa banyak node yang ditelusuri, runtime, dan pemakaian memory. Kemudian terdapat method displayPath, method ini yang akan menampilkan langkah dari awal hingga mencapai kata terakhir. Setiap huruf yang diubah akan ditandai dengan warna hijau sehingga untuk mempermudah dalam melihat perbedaan. Langkahnya akan ditampilkan secara vertikal jadi start word di paling atas dan langkahnya berurut ke bawah sampai end word. Jika tidak ada hasil maka akan keluar no solution found. Setiap menekan tombol solve, path hasil pencarian yang sebelumnya akan dihapus dan diganti baru dengan pencarian yang baru.

BAB VI
PENUTUP

6.1. Kesimpulan

Dari hasil pengujian yang dilakukan terhadap tiga algoritma pencarian dalam word ladder solver (Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A*) dapat disimpulkan beberapa hal. Pertama, algoritma UCS menghasilkan solusi path yang optimal tetapi memerlukan waktu eksekusi yang lebih lama dan memori yang lebih besar dibandingkan dengan dua algoritma lainnya. Hal ini disebabkan oleh pendekatannya yang exhaustive dalam menjelajahi semua kemungkinan node pada setiap level baru lanjut ke level berikutnya. Kedua, algoritma GBFS menghasilkan solusi path yang tidak selalu optimal tetapi kecepatan eksekusi yang sangat cepat karena fokusnya yang kuat pada heuristik serta pemakaian memory yang paling sedikit. Pendekatan ini seringkali tidak menghasilkan solusi yang optimal karena tidak memikirkan biaya dari awal yang menyebabkan solusi terjebak dalam optimum lokal. Terakhir, algoritma A* menunjukkan keseimbangan terbaik antara kecepatan dan efisiensi solusi, dengan menggabungkan biaya yang telah dikeluarkan dan estimasi heuristik ke tujuan untuk memprioritaskan pencarian solusi. Algoritma A* seringkali menghasilkan solusi yang optimal dengan waktu eksekusi yang lebih cepat dan penggunaan memori yang lebih sedikit dibandingkan dengan UCS.

Dalam konteks aplikasi word ladder solver, A* secara konsisten mengungguli UCS dalam hal kecepatan dan pemakaian memori dengan optimalitas path yang ditemukan. Hal ini menjadikannya pilihan yang sangat baik untuk aplikasi yang membutuhkan kombinasi antara efisiensi waktu dan penggunaan sumber daya. Algoritma GBFS dapat dijadikan pilihan untuk skenario dimana kecepatan adalah prioritas utama dan keakuratan atau minimalisasi langkah bukan hal yang terlalu dipentingkan. Algoritma UCS, meskipun kurang efisien dalam hal sumber daya dan waktu, tetap menjadi pilihan dalam kasus dimana solusi untuk menemukan jalur terpendek adalah hal yang diutamakan. Oleh karena itu, pemilihan algoritma dapat disesuaikan berdasarkan kebutuhan pengguna.

6.2. Saran

Untuk meningkatkan kinerja dan efisiensi dalam menyelesaikan permainan word ladder, Algoritma GBFS dan A* dapat dikembangkan dengan cara mengganti atau mengembangkan heuristik yang lebih akurat.

DAFTAR PUSTAKA

Munir, Rinaldi. Penentuan rute (Route/Path Planning) - Bagian 1. Diakses 3 Mei 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

Munir, Rinaldi. Penentuan rute (Route/Path Planning) - Bagian 2. Diakses 3 Mei 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf

LAMPIRAN

A. Pranala Repository: https://github.com/BocilBlunder/Tucil3_13522113.git

B. Checklist:

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	✔	