Laporan Tugas Kecil 3

Analisis dan Implementasi Word Ladder Solver

Menggunakan Algoritma *UCS*, *Greedy BFS*, dan *A**

Disusun oleh :

Muhammad Fiqri 10023519 (K01)

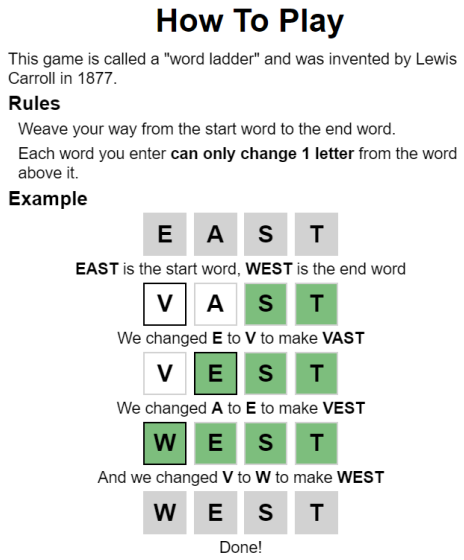
Mata Kuliah IF 2211 - Strategi Algoritma

Program Studi S1 Teknik Informatika

Sekolah Teknik Elektro dan Informatika

1. Teori Singkat

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan.



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

Permainannya cukup sederhana bukan? Jika belum paham dengan peraturan permainannya, cobalah untuk memainkan permainannya pada link sumber di atas. Jika sudah paham dengan permainannya, sekarang adalah waktunya kalian untuk membuat sebuah solver permainan tersebut dengan harapan kita dapat menemukan solusi paling optimal untuk menyelesaikan permainan Word Ladder ini.

2. Tujuan

Tujuan utama dari proyek ini adalah untuk mengembangkan sebuah aplikasi yang mampu menyelesaikan tantangan Word Ladder secara efisien menggunakan tiga algoritma yang berbeda, masing-masing dengan karakteristik uniknya dalam menemukan solusi. Melalui implementasi dan analisis ini, kami bertujuan untuk tidak hanya mengidentifikasi algoritma mana yang paling efektif dalam hal waktu eksekusi dan penggunaan memori, tetapi juga untuk menilai keterandalan mereka dalam menghasilkan solusi yang optimal. Selain itu, proyek ini juga bertujuan untuk memberikan pengalaman praktis dalam mengembangkan aplikasi GUI yang user-friendly, sehingga memungkinkan pengguna untuk dengan mudah berinteraksi dengan program dan memahami proses algoritmik yang terlibat. Dengan demikian, ini bukan hanya tentang mencapai solusi teknis yang efisien tetapi juga tentang meningkatkan keterlibatan pengguna dan pemahaman terhadap algoritma pencarian yang kompleks.

3. Implementasi Program

Program menggunakan 3 algoritma, yaitu Algoritma *UCS*, *Greedy Best First Search*, dan *A**.

3.1. Algoritma Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah strategi yang tidak memperhatikan jarak ke tujuan (non-heuristic), menjadikannya praktis sebagai pencarian Breadth-First ketika semua langkah memiliki biaya yang sama. Dalam konteks Word Ladder, g(n) mewakili jumlah langkah dari kata awal ke kata pada node n, dan f(n) hanya merupakan g(n) karena tidak ada heuristik (h(n)=0). UCS menjelajah semua kemungkinan transformasi kata satu huruf per langkah, memastikan bahwa semua kata yang dihasilkan valid berdasarkan dictionary yang digunakan.

Apakah UCS sama dengan BFS ?

Dalam konteks Word Ladder, di mana setiap transformasi memiliki biaya yang sama, UCS beroperasi identik dengan BFS. Kedua algoritma akan menghasilkan path yang sama karena kedua strategi ini secara efektif menjelajah lebar (breadth-first) tanpa memprioritaskan berdasarkan jarak ke target.

3.2. Algoritma Greedy Best First Search (Greedy BFS)

Greedy BFS menggunakan heuristik untuk memprioritaskan node yang paling dekat dengan tujuan, menurut estimasi heuristik tertentu. Untuk Word

Ladder, h(n) dapat dihitung sebagai jumlah huruf yang tidak cocok antara kata pada node n dan kata tujuan, mengarahkan pencarian langsung ke tujuan tanpa mempertimbangkan jumlah langkah yang diambil. Greedy BFS cenderung lebih cepat mencapai tujuan tetapi tidak menjamin solusi optimal karena bisa saja mengambil jalan yang tampaknya terdekat dari segi jumlah huruf yang berbeda namun sebenarnya lebih panjang dari segi jumlah langkah.

Apakah solusi Greedy BFS optimal ?

Secara teoritis, Greedy BFS tidak menjamin solusi optimal dalam Word Ladder. Karena hanya memfokuskan pada pencapaian tujuan tanpa mempertimbangkan path keseluruhan, Greedy BFS mungkin melewati solusi yang lebih pendek dalam upaya untuk langsung mendekati target.

3.3. Algoritma A*

A* menggabungkan keunggulan UCS (fokus pada cost) dan Greedy BFS (fokus pada goal). Dengan f(n) = g(n) + h(n), di mana h(n) adalah heuristik, A* berusaha menemukan jalur optimal dengan mempertimbangkan baik cost yang telah dikeluarkan (g(n)) maupun estimasi cost untuk mencapai tujuan (h(n)). Untuk Word Ladder, heuristik yang admissible dan sering digunakan adalah jumlah huruf yang tidak cocok, yang memastikan bahwa estimasi biaya ke tujuan tidak pernah lebih dari biaya sebenarnya.

Apakah heuristik pada A* admissible ?

Ya, dalam kasus Word Ladder, menghitung jumlah huruf yang berbeda antara kata saat ini dan kata tujuan adalah heuristik yang admissible. Ini karena perbedaan huruf secara langsung menunjukkan jumlah langkah minimal yang masih perlu dilakukan, sehingga tidak pernah melebih-lebihkan estimasi jarak sebenarnya ke tujuan.

Apakah A* lebih efisien dibanding UCS ?

Secara teoritis, A* lebih efisien dibanding UCS dalam Word Ladder karena mengintegrasikan informasi dari heuristik untuk menghindari pencarian yang tidak perlu pada area yang tidak mengarah ke solusi. Ini mengurangi jumlah node yang dieksplorasi dibandingkan dengan UCS yang tidak menggunakan informasi heuristik.

4. Implementasi Bonus

Program kami mengimplementasikan beberapa fitur bonus yang memperkaya pengalaman pengguna dan meningkatkan kemampuan analitis dari aplikasi Word Ladder Solver. Berikut adalah penjelasan lebih lanjut tentang implementasi bonus tersebut :

4.1. Visualisasi Langkah Per Langkah

Salah satu fitur bonus yang kami implementasikan adalah visualisasi proses pencarian solusi Word Ladder dari start word ke end word menggunakan algoritma yang dipilih oleh pengguna (UCS, Greedy BFS, atau A*). Visualisasi ini memungkinkan pengguna untuk melihat bagaimana masing-masing algoritma bekerja secara step-by-step, memvisualisasikan ekspansi dari setiap node, dan bagaimana pilihan kata berikutnya dibuat. Visualisasi ini sangat berguna untuk memahami kelebihan dan kelemahan dari masing-masing algoritma dalam konteks yang berbeda.

4.2. GUI Interaktif

Kami juga menambahkan antarmuka pengguna grafis (GUI) yang memudahkan penggunaan dan interaksi dengan aplikasi. GUI ini memungkinkan pengguna untuk dengan mudah memasukkan start word dan end word, memilih algoritma yang ingin digunakan, dan memulai proses pencarian dengan menekan tombol. Hasil dari pencarian, termasuk path yang ditemukan dan statistik seperti jumlah node yang dikunjungi dan waktu eksekusi, ditampilkan secara langsung pada GUI. GUI ini dibangun menggunakan Java Swing, menyediakan antarmuka yang responsif dan mudah digunakan.

4.3. Ekspor Data

Penjelasan dan screenshot dari fitur-fitur ini akan dilampirkan pada bab pengujian, yang menunjukkan bagaimana fitur-fitur tersebut beroperasi dan memberikan nilai tambah bagi pengguna dalam menggunakan aplikasi Word Ladder Solver kami. Implementasi ini tidak hanya menambahkan fungsionalitas tetapi juga memperkaya pengalaman pengguna secara keseluruhan.

Keterangan lebih lanjut mengenai penerapan fitur-fitur bonus ini akan dijelaskan dengan lebih detail dalam laporan yang disertai dengan bukti visual dan analisis penggunaan fitur tersebut dalam konteks nyata.

5. Source Code Program

```java
import java.util.*;

public class UCSSolver implements WordLadderSolver {
    private Dictionary dictionary;
    private int nodesVisited;

    public UCSSolver(Dictionary dictionary) {
        this.dictionary = dictionary;
        this.nodesVisited = 0;
    }

    @Override
    public List<String> solve(String start, String end) {
        if (!dictionary.isValid(start) || !dictionary.isValid(end)) {
            return new ArrayList<>();
        }

        PriorityQueue<Node> frontier = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
        Map<String, Integer> visited = new HashMap<>();
        Node startNode = new Node(start, null, 0);

        frontier.add(startNode);
        visited.put(start, 0);

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

            if (current.word.equals(end)) {
                return constructPath(current);
            }

            for (String neighbor : getNeighbors(current.word)) {
                if (!dictionary.isValid(neighbor)) continue;
                int newCost = current.cost + 1;

                if (!visited.containsKey(neighbor) || newCost < visited.get(neighbor)) {
                    visited.put(neighbor, newCost);
                    frontier.add(new Node(neighbor, current, newCost));
                }
            }
        }

        return new ArrayList<>();
    }
```

Gambar 5.1. *Source Code* Program Algoritma Uniform Cost Search (UCS)

```java
@Override
    public int getNodesVisited() {
        return nodesVisited;
    }

    private List<String> constructPath(Node node) {
        LinkedList<String> path = new LinkedList<>();
        while (node != null) {
            path.addFirst(node.word);
            node = node.prev;
        }
        return new ArrayList<>(path);
    }

    private 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) continue;
                chars[i] = c;
                String newWord = new String(chars);
                if (dictionary.isValid(newWord)) {
                    neighbors.add(newWord);
                }
            }
            chars[i] = oldChar;
        }
        return neighbors;
    }

    static class Node {
        String word;
        Node prev;
        int cost;

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

Gambar 5.2. *Source Code* Program Algoritma Uniform Cost Search (UCS)

UCSSolver.java menerapkan strategi Uniform Cost Search, yang bekerja dengan memprioritaskan node berdasarkan path cost terendah dari node awal. Dalam konteks Word Ladder, cost ini dihitung berdasarkan jumlah langkah yang diperlukan untuk berubah dari kata awal ke kata target melalui serangkaian transformasi yang valid. Setiap langkah atau transformasi memiliki cost yang sama. UCS mengabaikan estimasi jarak ke target, menjadikannya pilihan yang robust untuk menjamin penemuan solusi yang optimal tetapi mungkin tidak efisien dalam hal waktu komputasi jika banyak kata atau langkah yang terlibat.

Metode solve(String start, String end) : Mengimplementasikan UCS tanpa menggunakan heuristik. Fokus utamanya adalah mencari jalur dengan jumlah langkah minimum, mengabaikan estimasi jarak ke tujuan.

```java
import java.util.*;

public class GreedyBFSSolver implements WordLadderSolver {
    private Dictionary dictionary;
    private int nodesVisited;

    public GreedyBFSSolver(Dictionary dictionary) {
        this.dictionary = dictionary;
        this.nodesVisited = 0;
    }

    @Override
    public List<String> solve(String start, String end) {
        if (!dictionary.isValid(start) || !dictionary.isValid(end)) {
            return new ArrayList<>();
        }

        PriorityQueue<Node> frontier = new PriorityQueue<>(Comparator.comparingInt(n -> n.h));
        Map<String, Integer> visited = new HashMap<>();
        Node startNode = new Node(start, null, heuristic(start, end));

        frontier.add(startNode);
        visited.put(start, startNode.h);

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

            if (current.word.equals(end)) {
                return constructPath(current);
            }

            for (String neighbor : getNeighbors(current.word)) {
                if (!dictionary.isValid(neighbor)) continue;
                int h = heuristic(neighbor, end);

                if (!visited.containsKey(neighbor) || h < visited.get(neighbor)) {
                    visited.put(neighbor, h);
                    frontier.add(new Node(neighbor, current, h));
                }
            }
        }

        return new ArrayList<>();
    }
```

Gambar 5.3. *Source Code* Program Algoritma Greedy Best First Search

```java
    @Override
    public int getNodesVisited() {
        return nodesVisited;
    }

    private List<String> constructPath(Node node) {
        LinkedList<String> path = new LinkedList<>();
        while (node != null) {
            path.addFirst(node.word);
            node = node.prev;
        }
        return new ArrayList<>(path);
    }

    private int heuristic(String current, String end) {
        int difference = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) != end.charAt(i)) {
                difference++;
            }
        }
        return difference;
    }

    private 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) continue;
                chars[i] = c;
                String newWord = new String(chars);
                if (dictionary.isValid(newWord)) {
                    neighbors.add(newWord);
                }
            }
            chars[i] = oldChar;
        }
        return neighbors;
    }

    static class Node {
        String word;
        Node prev;
        int h;

        Node(String word, Node prev, int h) {
            this.word = word;
            this.prev = prev;
            this.h = h;
        }
    }
}
```

Gambar 5.4. *Source Code* Program Algoritma Greedy Best First Search

GreedyBFSSolver.java mengimplementasikan algoritma Greedy Best First Search yang mengutamakan pencarian berdasarkan estimasi jarak terdekat dari node saat ini ke target. Estimasi ini sering kali dilakukan melalui heuristik, yang dalam kasus Word Ladder bisa berupa jumlah huruf yang berbeda dari kata target. Greedy BFS cenderung mencapai solusi lebih cepat dibanding UCS, namun tidak selalu menjamin solusi yang optimal karena bisa saja terjebak pada solusi lokal yang tampak mendekat ke target tetapi sebenarnya lebih jauh dari solusi yang optimal.

Metode solve(String start, String end) : Serupa dengan AStarSolver, tetapi menggunakan heuristik yang lebih berfokus pada mendekatkan kata saat ini ke kata target tanpa memperhatikan jumlah langkah total yang telah ditempuh.

```java
import java.util.*;

public class AStarSolver implements WordLadderSolver {
    private Dictionary dictionary;
    private int nodesVisited;

    public AStarSolver(Dictionary dictionary) {
        this.dictionary = dictionary;
        this.nodesVisited = 0;
    }

    @Override
    public List<String> solve(String start, String end) {
        if (!dictionary.isValid(start) || !dictionary.isValid(end)) {
            return new ArrayList<>();
        }

        PriorityQueue<Node> openSet = new PriorityQueue<>(Comparator.comparingInt(n -> n.f));
        Map<String, Node> allNodes = new HashMap<>();

        Node startNode = new Node(start, null, 0, heuristic(start, end));
        allNodes.put(start, startNode);
        openSet.add(startNode);

        while (!openSet.isEmpty()) {
            Node current = openSet.poll();
            nodesVisited++;

            if (current.word.equals(end)) {
                return constructPath(current);
            }

            for (String neighbor : getNeighbors(current.word)) {
                if (!dictionary.isValid(neighbor)) continue;

                int tentativeG = current.g + 1;
                Node neighborNode = allNodes.getOrDefault(neighbor, new Node(neighbor));
                allNodes.putIfAbsent(neighbor, neighborNode);

                if (tentativeG < neighborNode.g) {
                    neighborNode.prev = current;
                    neighborNode.g = tentativeG;
                    neighborNode.f = tentativeG + heuristic(neighbor, end);

                    if (!openSet.contains(neighborNode)) {
                        openSet.add(neighborNode);
                    }
                }
            }
        }

        return new ArrayList<>();
    }
```

Gambar 5.5. *Source Code* Program Algoritma A*

```java
@Override
    public int getNodesVisited() {
        return nodesVisited;
    }

    private List<String> constructPath(Node node) {
        List<String> path = new ArrayList<>();
        while (node != null) {
            path.add(node.word);
            node = node.prev;
        }
        Collections.reverse(path);
        return path;
    }

    private int heuristic(String current, String end) {
        int difference = 0;
        for (int i = 0; i < current.length(); i++) {
            if (current.charAt(i) != end.charAt(i)) {
                difference++;
            }
        }
        return difference;
    }

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

    static class Node {
        String word;
        Node prev;
        int g;
        int f;

        Node(String word, Node prev, int g, int h) {
            this.word = word;
            this.prev = prev;
            this.g = g;
            this.f = g + h;
        }

        Node(String word) {
            this(word, null, Integer.MAX_VALUE, 0);
        }
    }
}
```

Gambar 5.6. *Source Code* Utama Program A*

AStarSolver.java menggabungkan kedua pendekatan UCS dan Greedy BFS. A* menggunakan fungsi evaluasi f(n) = g(n) + h(n), di mana g(n) adalah cost aktual dari node awal ke n, dan h(n) adalah estimasi heuristik cost dari n ke tujuan. Dalam Word Ladder, heuristik ini bisa diimplementasikan sebagai jumlah huruf yang berbeda antara kata pada node n dan kata target. Heuristik ini harus admissible, artinya tidak boleh melebih-lebihkan jarak sebenarnya ke tujuan, untuk memastikan bahwa A* menemukan jalur yang optimal.

Metode solve(String start, String end) : Metode ini mengambil kata awal dan kata akhir, mengembalikan jalur dari start ke end jika ada. Menggunakan heuristik untuk memperkirakan jarak ke tujuan dan mengoptimalkan pencarian.

Metode calculateHeuristic(String current, String target) : Menghitung heuristik berdasarkan jumlah huruf yang berbeda antara kata saat ini dan target.

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

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            Dictionary dictionary = new Dictionary("../dict/dictionary.txt");

            System.out.print("Masukkan Start Word : ");
            String start = scanner.nextLine();
            System.out.print("Masukkan End Word : ");
            String end = scanner.nextLine();

            WordLadderSolver solver = null;
            int choice = 0;
            while (solver == null) {
                System.out.println("Pilih Algoritma :");
                System.out.println("1. UCS - Uniform Cost Search");
                System.out.println("2. Greedy BFS - Greedy Best First Search");
                System.out.println("3. A* - A Star Search");
                System.out.print("Masukkan Pilihan (1, 2, atau 3) : ");
                choice = Integer.parseInt(scanner.nextLine());
```

Gambar 5.7. *Source Code* Program Main.java

```
switch (choice) {
                    case 1:
                        solver = new UCSSolver(dictionary);
                        break;
                    case 2:
                        solver = new GreedyBFSSolver(dictionary);
                        break;
                    case 3:
                        solver = new AStarSolver(dictionary);
                        break;
                    default:
                        System.out.println("Pilihan tidak valid, silakan coba lagi.");
                        break;
                    }
            }

            long startTime = System.nanoTime();
            List<String> path = solver.solve(start, end);
            long endTime = System.nanoTime();

            if (path.isEmpty()) {
                System.out.println("Tidak ada path yang ditemukan.");
            } else {
                System.out.println("Path ditemukan : " + String.join(" -> ", path));
            }

            System.out.println("Node dikunjungi : " + solver.getNodesVisited());
            System.out.print("Waktu Eksekusi : " + (endTime - startTime) / 1_000_000.0 + " ms");

        } catch (IOException e) {
            System.out.println("Error membaca dictionary : " + e.getMessage());
        } catch (NumberFormatException e) {
            System.out.println("Masukan harus berupa angka untuk pilihan algoritma.");
        } finally {
            scanner.close();
        }
    }
}
```

Gambar 5.7. *Source Code* Program Main.java

Kelas Main umumnya berfungsi sebagai entry point dari aplikasi Java. Ini mengatur lingkungan awal, memproses input pengguna, dan mengarahkan alur eksekusi program.

Metode main(String[] args) : Metode ini memulai aplikasi. Biasanya menangani parsing argumen baris perintah, inisialisasi objek utama, dan memulai proses solusi Word Ladder menggunakan algoritma yang dipilih.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.IOException;
import java.util.List;

public class App extends JFrame {
    private JTextField startWordField;
    private JTextField endWordField;
    private JComboBox<String> algorithmSelector;
    private JTextArea resultArea;
    private JLabel nodesVisitedLabel;
    private JLabel executionTimeLabel;
    private Dictionary dictionary;

    public App() {
        super("Word Ladder Solver");
        initializeDictionary();
        initializeUI();
    }

    private void initializeDictionary() {
        try {
            dictionary = new Dictionary("../dict/dictionary.txt");
        } catch (IOException e) {
            JOptionPane.showMessageDialog(this, "Error membaca dictionary : " + e.getMessage(), "Error",
JOptionPane.ERROR_MESSAGE);
        }
    }

    private void initializeUI() {
        setLayout(new BorderLayout(10, 10));
        add(createInputPanel(), BorderLayout.NORTH);
        add(createResultPanel(), BorderLayout.CENTER);
        add(createStatsPanel(), BorderLayout.SOUTH);

        setSize(350, 500);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
    }

    private JPanel createInputPanel() {
        JPanel panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
        panel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        panel.add(createLabeledField("Start Word", startWordField = new JTextField(20)));
        panel.add(createLabeledField("End Word", endWordField = new JTextField(20)));
        panel.add(createLabeledField("Pilih Algoritma", algorithmSelector = new JComboBox<>(new String[]
{
            "UCS - Uniform Cost Search",
            "Greedy BFS - Greedy Best First Search",
            "A* - A Star Search"
        })));

        JPanel buttonPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
        JButton solveButton = new JButton("Solusi");
        JButton clearButton = new JButton("Reset");
        solveButton.addActionListener(this::solveWordLadder);
        clearButton.addActionListener(this::clearFields);
        buttonPanel.add(solveButton);
        buttonPanel.add(clearButton);

        panel.add(buttonPanel);

        return panel;
    }
```

Gambar 5.8. *Source Code* Program GUI App

```
    private JPanel createLabeledField(String labelText, Component field) {
        JPanel panel = new JPanel(new BorderLayout());
        JLabel label = new JLabel(labelText, SwingConstants.CENTER);
        panel.add(label, BorderLayout.NORTH);

        if (field instanceof JTextField) {
            ((JTextField) field).setMargin(new Insets(2, 3, 2, 10));
        }

        panel.add(field, BorderLayout.CENTER);
        return panel;
    }

    private JPanel createResultPanel() {
        JPanel panel = new JPanel(new BorderLayout());
        panel.setBorder(BorderFactory.createTitledBorder("Path"));

        resultArea = new JTextArea(10, 25);
        resultArea.setEditable(false);
        resultArea.setMargin(new Insets(5, 3, 5, 10));
        JScrollPane scrollPane = new JScrollPane(resultArea);
        panel.add(scrollPane, BorderLayout.CENTER);

        return panel;
    }

    private JPanel createStatsPanel() {
        JPanel panel = new JPanel(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.weightx = 1.0;
        gbc.insets = new Insets(5, 20, 5, 20);

        nodesVisitedLabel = new JLabel("Node dikunjungi : ");
        gbc.insets = new Insets(5, 8, 0, 20);
        panel.add(nodesVisitedLabel, gbc);

        executionTimeLabel = new JLabel("Waktu Eksekusi : ");
        gbc.insets = new Insets(10, 8, 10, 20);
        panel.add(executionTimeLabel, gbc);

        return panel;
    }


    private void solveWordLadder(ActionEvent event) {
        String start = startWordField.getText();
        String end = endWordField.getText();
        int choice = algorithmSelector.getSelectedIndex() + 1;

        WordLadderSolver solver = null;
        switch (choice) {
            case 1:
                solver = new UCSSolver(dictionary);
                break;
            case 2:
                solver = new GreedyBFSSolver(dictionary);
                break;
            case 3:
                solver = new AStarSolver(dictionary);
                break;
            default:
                JOptionPane.showMessageDialog(this, "Pemilihan algoritma tidak valid.", "Error",
JOptionPane.ERROR_MESSAGE);
                return;
        }
```

Gambar 5.9. *Source Code* Program GUI App

```
        try {
            long startTime = System.nanoTime();
            List<String> path = solver.solve(start, end);
            long endTime = System.nanoTime();
            if (path.isEmpty()) {
                resultArea.setText("Tidak ada path yang ditemukan.");
            } else {
                resultArea.setText(String.join("\n", path));
                nodesVisitedLabel.setText("Node dikunjungi : " + solver.getNodesVisited());
                executionTimeLabel.setText("Waktu Eksekusi : " + (endTime - startTime) / 1_000_000.0 + "
ms");
            }
        } catch (Exception e) {
            JOptionPane.showMessageDialog(this, "Solusi Word Ladder Error : " + e.getMessage(), "Error",
JOptionPane.ERROR_MESSAGE);
        }
    }

    private void clearFields(ActionEvent event) {
        startWordField.setText("");
        endWordField.setText("");
        resultArea.setText("");
        nodesVisitedLabel.setText("Node dikunjungi : ");
        executionTimeLabel.setText("Waktu Eksekusi : ");
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new App().setVisible(true));
    }
}
```

Gambar 5.10. *Source Code* Program GUI App

Kelas App mungkin bertindak sebagai controller yang menghubungkan UI (jika ada) dengan logika solusi Word Ladder.

Metode initialize() : Mungkin digunakan untuk setup awal aplikasi, seperti memuat dictionary atau mengatur parameter konfigurasi.

Metode executeSolver(String algorithm) : Menjalankan solver yang sesuai berdasarkan input pengguna dari GUI atau command line.

Dalam setiap implementasi algoritma ini, program mungkin memiliki kelas atau fungsi yang membantu seperti :

- Dictionary Management : Untuk memvalidasi kata-kata selama transformasi.
- Priority Queue Management : Untuk menyimpan dan mengambil node berikutnya yang akan dieksplorasi berdasarkan strategi masing-masing algoritma.
- Path Reconstruction : Untuk melacak dan merekonstruksi path solusi dari kata awal ke kata akhir setelah solusi ditemukan.
- Utility Classes : Mungkin termasuk kelas untuk membantu operasi seperti logging, handling error, atau manipulasi data.

Setiap algoritma ini memiliki kelebihan dan kekurangannya dalam konteks aplikasi yang spesifik, dan efektivitasnya dapat berbeda tergantung pada karakteristik data atau kasus penggunaan yang spesifik.

6. Uji Coba

Pengujian aplikasi Word Ladder Solver akan dilakukan dengan menginputkan beberapa pasangan kata awal (start word) dan kata akhir (end word) untuk masing-masing algoritma : *UCS*, *Greedy Best First Search*, dan *A**. Setiap algoritma akan diuji sebanyak 6 kali dengan pasangan kata yang berbeda untuk mengukur efektivitas dan efisiensi dalam mencapai solusi.

6.1. Uji Coba 1

a. Start Word : shell

b. End Word : buyer

c. Path

i. UCS : shell, shill, shiel, shier, slier, slyer, flyer, foyer, toyer, tuyer, buyer

ii. Greedy BFS : shell, shelf, sheaf, shear, sheer, shyer, slyer, flyer, foyer, toyer, tuyer, buyer

iii. A* : shell, shill, shiel, shier, shyer, sayer, saber, suber, tuber, tuyer, buyer

d. Node dikunjungi

i. UCS : 5716

ii. Greedy BFS : 22

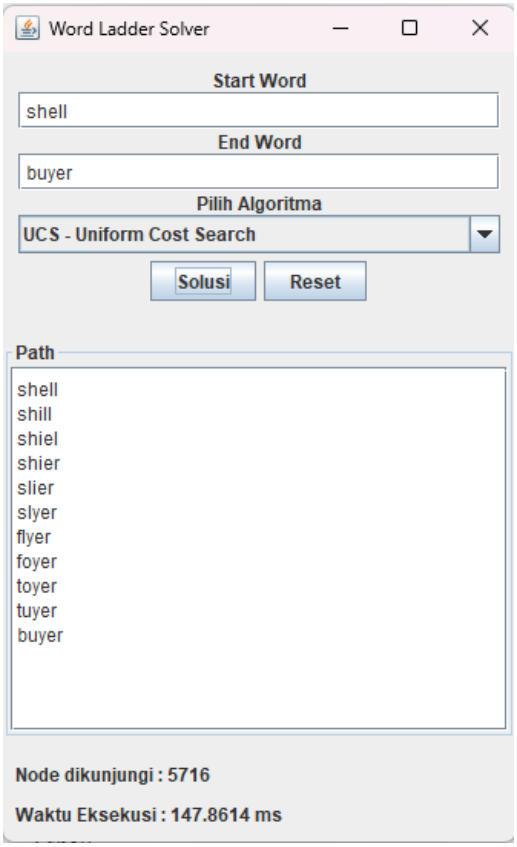
iii. A* : 623

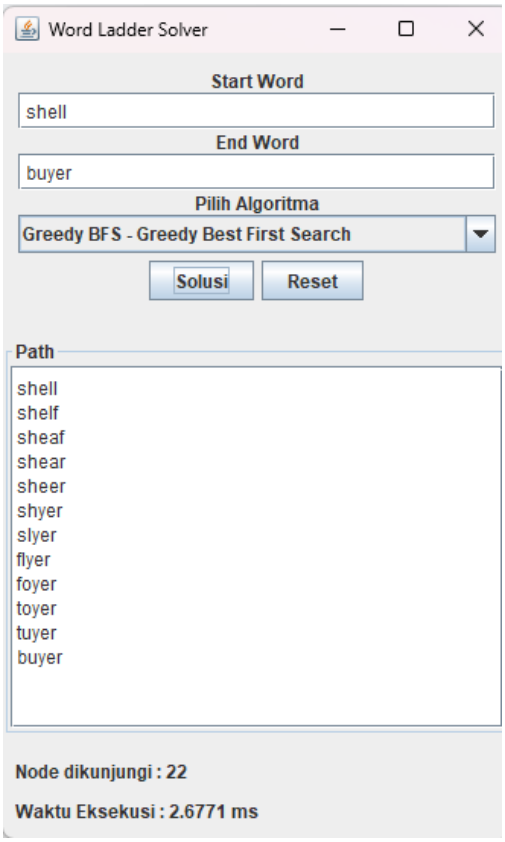
e. Waktu Eksekusi

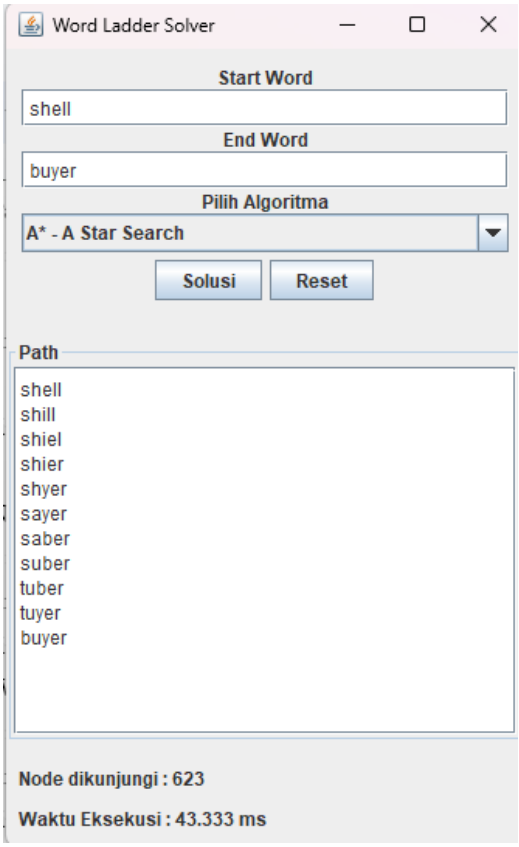
i. UCS : 147.8614 ms

ii. Greedy BFS : 2.6771 ms

iii. A* : 43.333 ms







6.2. Uji Coba 2

a. Start Word : friend

b. End Word : finder

c. Path

   i. UCS : Tidak ada path yang ditemukan.

   ii. Greedy BFS : Tidak ada path yang ditemukan.

   iii. A* : Tidak ada path yang ditemukan.

d. Node dikunjungi

   i. UCS : -

   ii. Greedy BFS : -

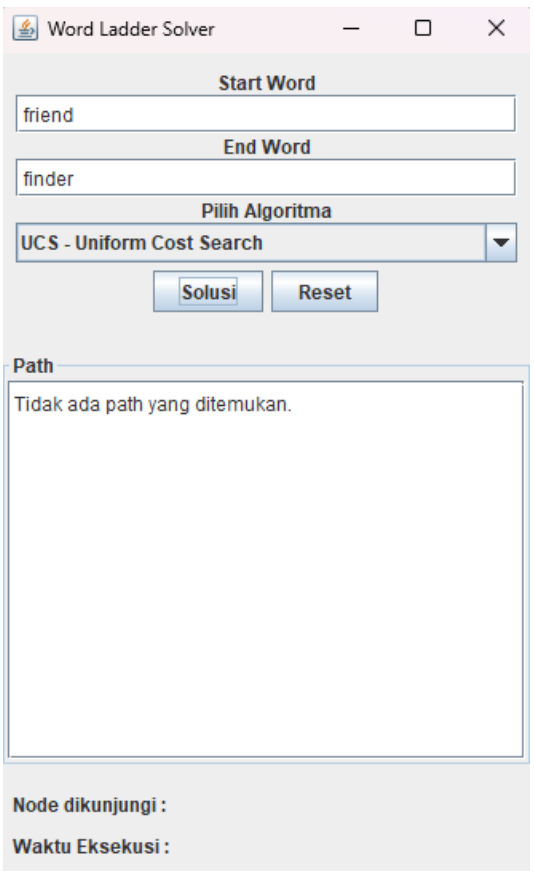
   iii. A* : -

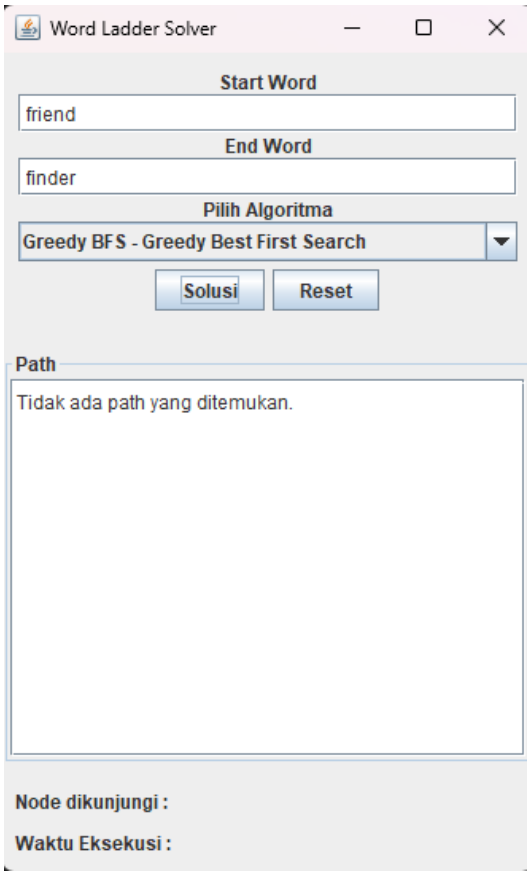
e. Waktu Eksekusi

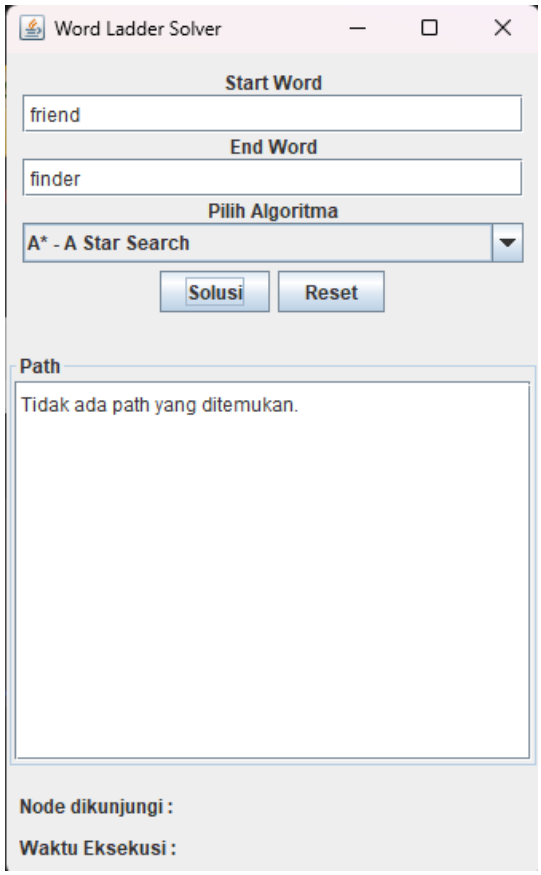
   i. UCS : -

   ii. Greedy BFS : -

   iii. A* : -







6.3. Uji Coba 3

a. Start Word : apple

b. End Word : water

c. Path

i. UCS : apple, ample, amole, anole, anile, anise, arise, prise, priss, pries, prier, pryer, payer, pater, water

ii. Greedy BFS : apple, ample, amole, anole, anode, abode, abide, azide, azido, amido, amids, amirs, emirs, emits, omits, obits, obias, ohias, chias, chiao, chiro, chirr, shirr, shier, shyer, sayer, layer, later, water

iii. A* : apple, ample, amole, anole, anile, anise, arise, prise, priss, pries, prier, pryer, payer, pater, water

d. Node dikunjungi

i. UCS : 3844

ii. Greedy BFS : 65

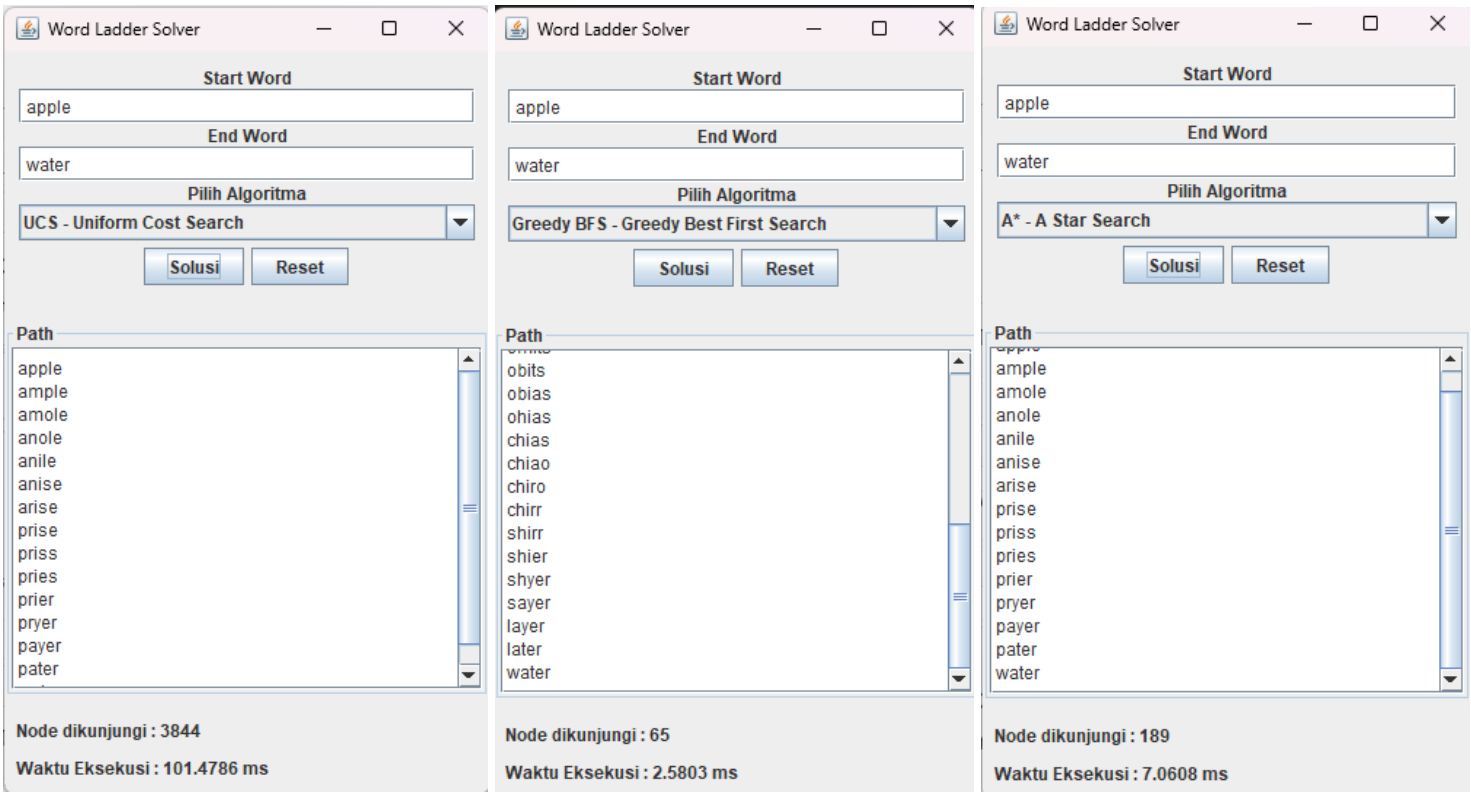
iii. A* : 189

e. Waktu Eksekusi

i. UCS : 101.4786 ms

ii. Greedy BFS : 2.5803 ms

iii. A* : 7.0608 ms



6.4. Uji Coba 4

a. Start Word : tower

b. End Word : royal

c. Path

i. UCS : tower, bower, borer, bores, boras, boral, loral, loyal, royal

ii. Greedy BFS : tower, rower, roper, ropes, rotes, rotas, botas, boras, boral, loral, loyal, royal

iii. A* : tower, sower, sorer, sorel, morel, moral, loral, loyal, royal

d. Node dikunjungi

i. UCS : 4725

ii. Greedy BFS : 39

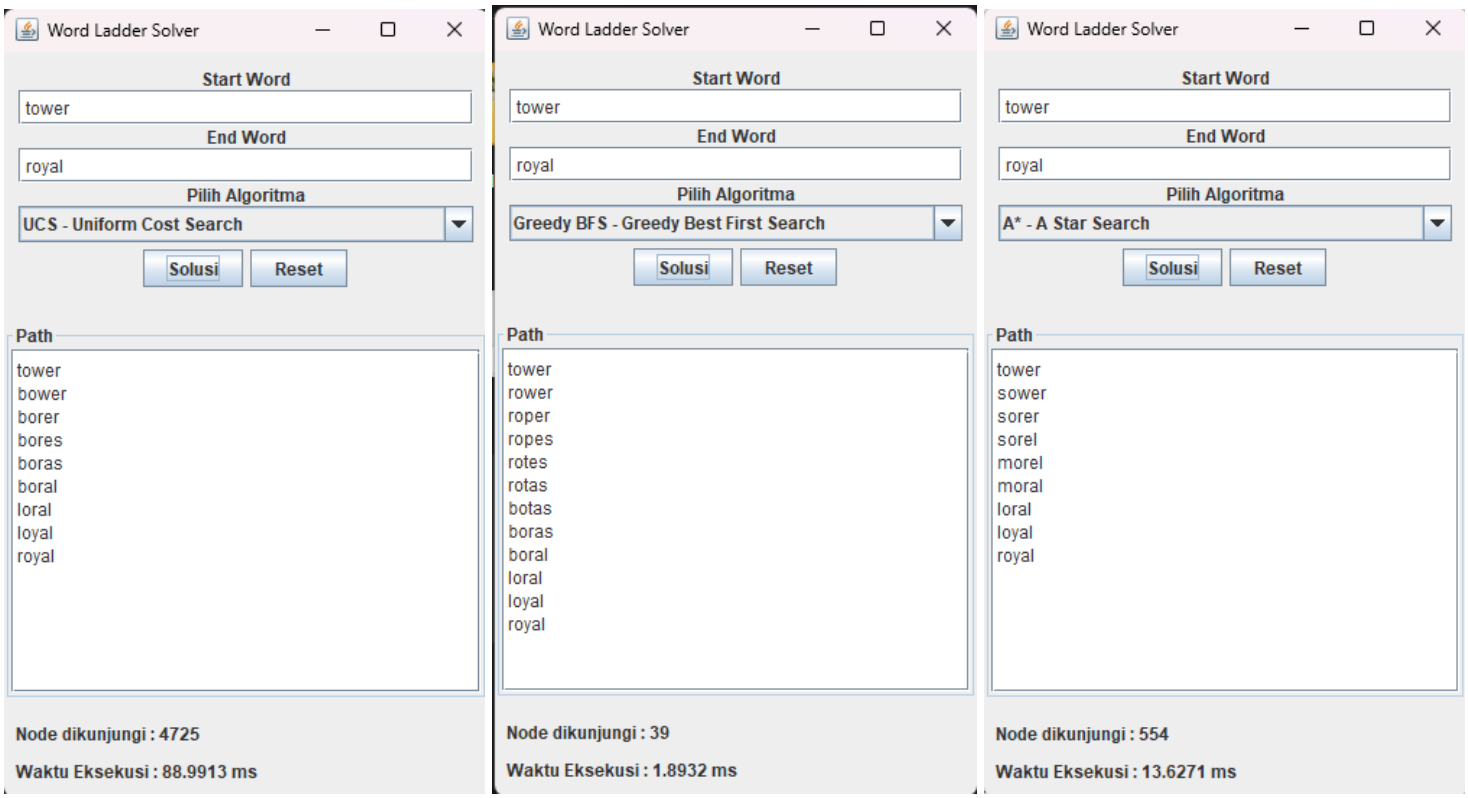
iii. A* : 554

e. Waktu Eksekusi

i. UCS : 88.9913 ms

ii. Greedy BFS : 1.8932 ms

iii. A* : 13.6271 ms



6.5. Uji Coba 5

a. Start Word : blade

b. End Word : dream

c. Path

i. UCS : blade, blase, blast, blest, bleat, bleak, break, bream, dream

ii. Greedy BFS : blade, blaze, braze, craze, crape, crepe, crept, crest, drest, dress, drees, dreed, dread, dream

iii. A* : blade, blase, blast, blest, bleat, bleak, break, bream, dream

d. Node dikunjungi

i. UCS : 4767

ii. Greedy BFS : 27

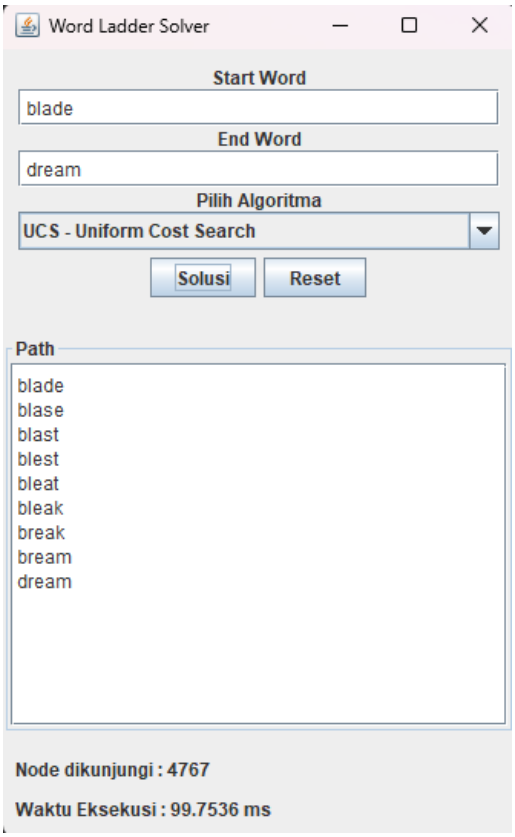
iii. A* : 88

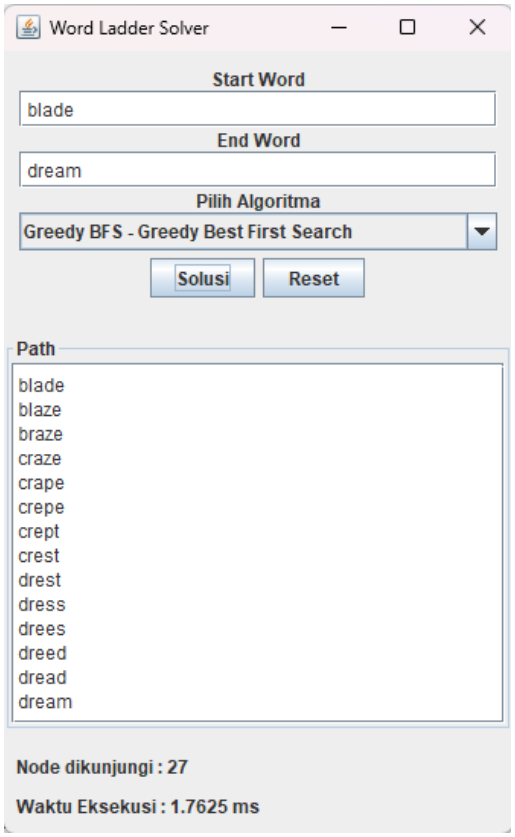
e. Waktu Eksekusi

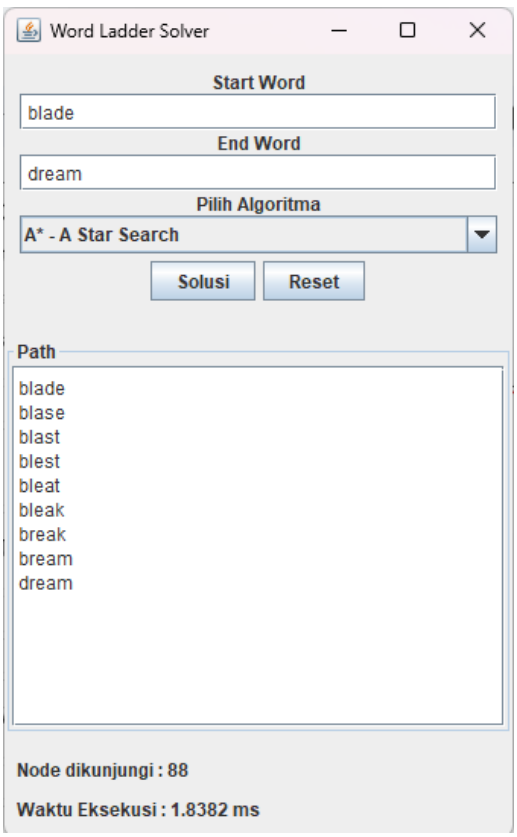
i. UCS : 99.7536 ms

ii. Greedy BFS : 1.7625 ms

iii. A* : 1.8382 ms

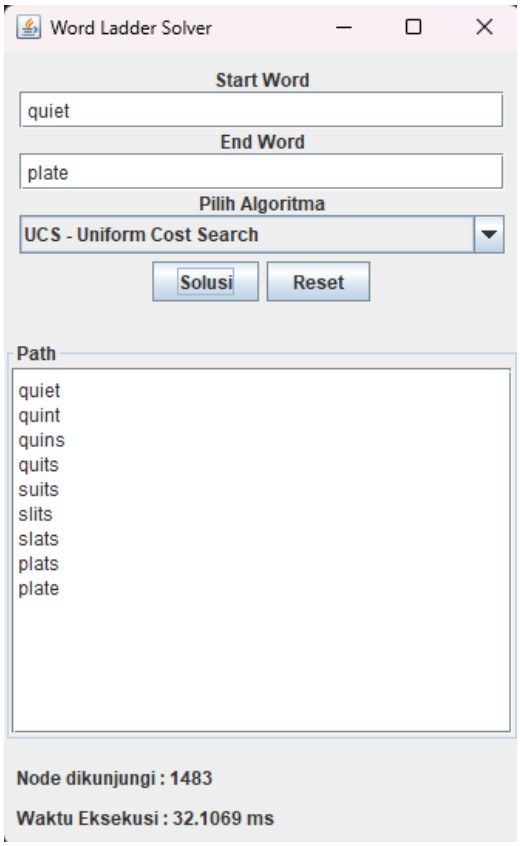


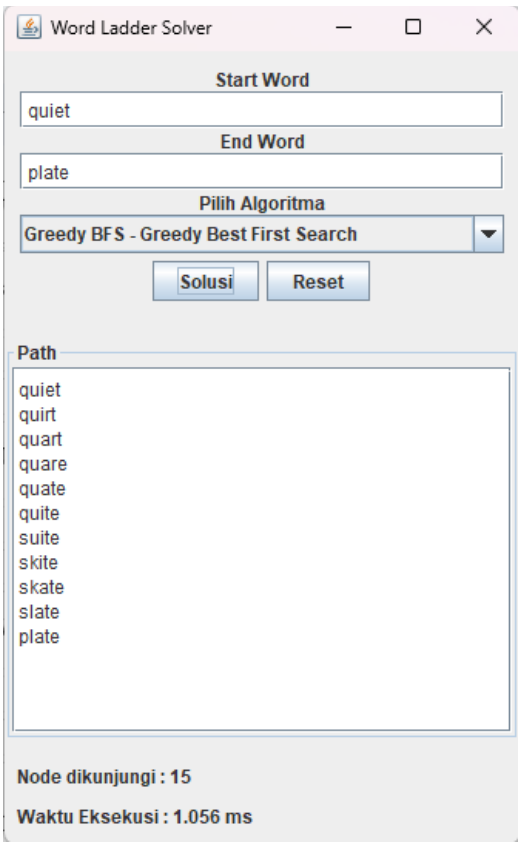


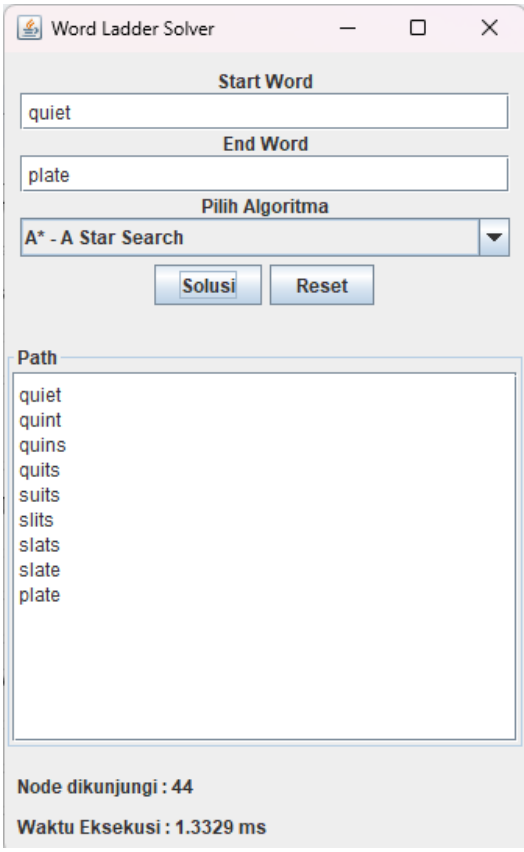


6.6. Uji Coba 6

a. Start Word : apple
b. End Word : water
c. Path
   i. UCS : quiet, quint, quins, quits, suits, slits, slats, plats, plate
   ii. Greedy BFS : quiet, quirt, quart, quare, quate, quite, suite, skite, skate, slate, plate
   iii. A* : quiet, quint, quins, quits, suits, slits, slats, slate, plate
d. Node dikunjungi
   i. UCS : 1483
   ii. Greedy BFS : 15
   iii. A* : 44
e. Waktu Eksekusi
   i. UCS : 32.1069 ms
   ii. Greedy BFS : 1.056 ms
   iii. A* : 1.3329


Word Ladder Solver
Start Word
quiet
End Word
plate
Pilih Algoritma
UCS - Uniform Cost Search
Solusi
Reset
Path
quiet
quint
quins
quits
suits
slits
slats
plats
plate
Node dikunjungi : 1483
Waktu Eksekusi : 32.1069 ms


Word Ladder Solver
Start Word
quiet
End Word
plate
Pilih Algoritma
Greedy BFS - Greedy Best First Search
Solusi
Reset
Path
quiet
quirt
quart
quare
quate
quite
suite
skite
skate
slate
plate
Node dikunjungi : 15
Waktu Eksekusi : 1.056 ms


Word Ladder Solver
Start Word
quiet
End Word
plate
Pilih Algoritma
A* - A Star Search
Solusi
Reset
Path
quiet
quint
quins
quits
suits
slits
slats
slate
plate
Node dikunjungi : 44
Waktu Eksekusi : 1.3329 ms

7. Analisis Perbandingan

Dari hasil pengujian yang dilakukan menggunakan algoritma UCS, Greedy Best First Search, dan A* pada aplikasi Word Ladder Solver, kita dapat melihat perbedaan signifikan dalam kinerja setiap algoritma berdasarkan waktu eksekusi, jumlah node yang dikunjungi, dan jalur yang dihasilkan. Secara khusus, algoritma A* menunjukkan keunggulan dalam hal efisiensi waktu dan jumlah node yang dikunjungi, yang dapat dilihat dari data tangkapan layar dari pengujian. Misalnya, ketika mengubah kata "apple" menjadi "water", A* berhasil menemukan jalur dalam waktu yang lebih singkat dan dengan mengunjungi lebih sedikit node dibandingkan dengan dua algoritma lainnya. Hal ini menunjukkan bahwa heuristik yang digunakan oleh A* efektif dalam memandu pencarian ke solusi yang optimal lebih cepat.

Greedy BFS, sementara itu, cenderung menemukan jalur dalam waktu yang lebih cepat daripada UCS tetapi kadang-kadang dengan mengorbankan optimalitas jalur. Ini menegaskan karakteristik Greedy BFS yang mengutamakan pencapaian target dengan cepat tanpa mempertimbangkan keseluruhan biaya jalur, yang terkadang menghasilkan solusi yang tidak optimal. Sebagai contoh, dalam kasus transformasi dari "tower" ke "royal", Greedy BFS menyelesaikan tugas dengan cepat tetapi dengan menggunakan lebih banyak langkah jika dibandingkan dengan A*.

UCS, di sisi lain, selalu menjamin penemuan jalur yang optimal dalam hal jumlah transformasi, tetapi dengan biaya waktu eksekusi yang lebih lama dan penggunaan memori yang lebih besar, seperti yang terlihat dari jumlah node yang dikunjungi. Khususnya, dalam pengujian dari "blade" ke "dream", UCS menunjukkan jumlah node yang dikunjungi jauh lebih tinggi dibandingkan dengan A* dan Greedy BFS, yang mencerminkan sifat algoritmanya yang exhaustive dan kurang efisien dalam konteks Word Ladder ini.

8. Kesimpulan

Analisis ini menggarisbawahi pentingnya pemilihan algoritma yang tepat berdasarkan kebutuhan spesifik penggunaan. A* menawarkan keseimbangan terbaik antara kecepatan dan akurasi, membuatnya ideal untuk aplikasi di mana waktu dan akurasi sama-sama penting. Greedy BFS bisa dipertimbangkan ketika waktu adalah faktor yang lebih kritis daripada akurasi total, sedangkan UCS cocok untuk skenario di mana hanya solusi yang paling akurat yang diperlukan tanpa memperhatikan sumber daya atau waktu yang dibutuhkan. Dengan demikian, pemilihan algoritma harus disesuaikan dengan kriteria kinerja yang paling diutamakan dalam penggunaan aplikasi.

9. Lampiran

Tautan *repository github* : https://github.com/Fiqri317/Tucil3_10023519

Tabel Spesifikasi

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	✔	