

LAPORAN TUGAS KECIL 03
IF2211 STRATEGI ALGORITMA

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



Disusun oleh:
Filbert 13522021

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

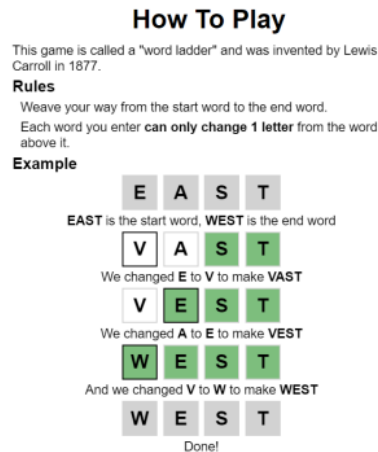
2024

DAFTAR ISI

BAB I	
DESKRIPSI MASALAH.....	2
BAB II	
ALGORITMA UCS, GREEDY BEST FIRST SEARCH, A* UNTUK PENYELESAIAN PERMAINAN WORD LADDER.....	3
2.1 Algoritma Uniform Cost Search (UCS).....	3
2.2 Algoritma Greedy Best First Search.....	5
2.3 Algoritma A* Search.....	6
BAB III	
IMPLEMENTASI PROGRAM.....	9
3.1. Main.java.....	9
3.2. GUI.java (Bonus).....	11
3.3. SearchResults.java.....	14
3.4. WordLoader.java.....	15
3.5. UCS.java.....	15
3.6. GBFS.java.....	17
3.7. Astar.java.....	19
3.8. NeighborGenerator.java.....	21
BAB IV	
ANALISIS DAN PENGUJIAN.....	23
4.1. Pengujian 1.....	23
4.1.1 Pengujian Algoritma Uniform Cost Search.....	23
4.1.2 Pengujian Algoritma Greedy Best First Search.....	27
4.1.3 Pengujian Algoritma A* Search.....	30
4.2. Analisis.....	34
4.3. Penjelasan Bonus.....	38
BAB V	
PENUTUP.....	39
5.1. Kesimpulan.....	39
5.2. Saran.....	40
DAFTAR REFERENSI.....	41
LAMPIRAN.....	42

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.

BAB II

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

2.1 Algoritma Uniform Cost Search (UCS)

Uniform Cost Search (UCS) merupakan algoritma pencarian yang digunakan untuk menemukan jalur terpendek dengan biaya terendah antara titik awal dan titik tujuan. Algoritma ini merupakan bentuk khusus dari algoritma pencarian *Dijkstra* yang dimodifikasi untuk tujuan pencarian jalur. UCS menggunakan struktur data *priority queue*, atau *heap*, untuk menyimpan semua simpul yang menunggu untuk dieksplorasi, dengan 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 yang direpresentasikan oleh fungsi $g(n)$. 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 dan memperbarui nilai $g(n)$ yang lebih rendah. Hal ini memastikan bahwa nilai $g(n)$ yang terkait dengan setiap simpul terus diperbarui secara dinamis selama pencarian berlangsung, sehingga jalur yang ditemukan pada akhirnya akan merupakan jalur dengan biaya terendah.

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

1. Algoritma dimulai dengan menentukan kata awal dan kata tujuan. Validasi dilakukan untuk memastikan bahwa kedua kata memiliki panjang yang sama, jika ya, pencarian dihentikan dengan segera karena solusi sudah ditemukan. Jika tidak, algoritma menyiapkan struktur data seperti *priority queue* yang digunakan untuk menyimpan simpul yang akan dieksplorasi, diurutkan berdasarkan cost. Simpul disimpan dalam bentuk objek *Node* yang menyimpan kata, simpul induk (*parent*), dan cost. Sebuah *hashmap visited*

digunakan untuk melacak kata-kata yang sudah dikunjungi dan cost terkecil untuk mencapainya.

2. Algoritma mengeluarkan simpul dengan cost terendah dari priority queue untuk dieksplorasi. Jika simpul tersebut adalah kata tujuan, maka jalur dari kata awal ke kata tujuan direkonstruksi dan dikembalikan. Jika bukan, algoritma menghasilkan semua tetangga valid dari kata tersebut—kata-kata yang bisa dicapai dengan mengubah satu huruf pada satu waktu.

3. Untuk setiap tetangga yang dihasilkan, algoritma menghitung cost baru (cost sebelumnya ditambah satu karena satu perubahan huruf). Jika tetangga belum dikunjungi atau memiliki cost yang lebih rendah daripada yang tercatat sebelumnya, maka tetangga tersebut dimasukkan ke dalam priority queue dengan cost baru dan diupdate dalam hashmap visited.

4. Proses mengeluarkan simpul, eksplorasi, dan pengembangan diulangi hingga priority queue kosong, yang berarti tidak ada jalur yang mungkin dari kata awal ke kata akhir, atau hingga jalur ke kata tujuan ditemukan.

5. Jika hasil sudah ditemukan, Algoritma mengembalikan hasil berupa jalur dari kata awal ke kata akhir dan jumlah simpul yang dikunjungi selama pencarian.

Metode UCS ini memastikan bahwa solusi yang ditemukan adalah optimal dalam hal jumlah perubahan huruf dari kata awal ke kata akhir, karena selalu memprioritaskan simpul dengan cost terkecil untuk dieksplorasi terlebih dahulu. Selain itu, penggunaan hashmap visited untuk menyimpan cost memastikan bahwa kata yang sama tidak dieksplorasi lebih dari satu kali dengan cost yang lebih tinggi, mempercepat pencarian dan mengurangi penggunaan memori.

Dalam UCS, cost berfungsi sebagai penentu prioritas eksplorasi simpul. Setiap simpul diwakili oleh sebuah kata, dan setiap perubahan huruf yang menghasilkan kata baru dianggap sebagai sebuah langkah dengan cost tambahan satu. Oleh karena itu, algoritma menghitung cost total untuk setiap kata yang dihasilkan sebagai jumlah langkah dari kata awal. Simpul dengan cost terendah selalu diutamakan, yang memastikan bahwa algoritma tidak hanya mencapai tujuan tetapi melakukannya dengan jumlah langkah yang paling efisien.

Proses ini sangat berguna dalam situasi di mana beberapa rute mungkin ada untuk mencapai kata tujuan. Misalnya, jika ada dua rute untuk berubah dari "cat" ke "dog", satu melalui "cot" dengan total 5 perubahan huruf dan yang lain melalui "cog"

dengan 4 perubahan, UCS akan memilih rute melalui "cog" karena memiliki cost lebih rendah (lebih sedikit langkah). Ini menunjukkan bagaimana UCS secara efektif mengevaluasi setiap kemungkinan rute berdasarkan metrik yang kuantitatif, yaitu cost, sehingga memastikan bahwa solusi optimal selalu dikejar.

2.2 Algoritma Greedy Best First Search

Greedy Best-First Search (GBFS) adalah algoritma yang menggunakan heuristik untuk mengarahkan pencarian ke tujuan dengan cara yang paling efisien menurut perkiraan algoritma. Dalam *GBFS*, fokus utama adalah pada pencarian yang cepat ke tujuan tanpa memperhitungkan total biaya jalur. Ini berarti algoritma selalu memilih untuk memperluas simpul yang tampaknya paling dekat dengan tujuan berdasarkan estimasi heuristik, tanpa mempertimbangkan biaya total dari simpul awal ke simpul saat ini.

Heuristik dalam *GBFS* harus memilih simpul yang secara intuitif dan secara geografis dekat dengan tujuan, seperti menggunakan jarak Euclidean dalam masalah jalur terpendek pada peta atau menggunakan hamming distance dalam masalah pencarian kata. Namun, karena algoritma ini mengabaikan biaya yang telah dikeluarkan untuk mencapai simpul saat ini, sering kali bisa terjebak oleh penghalang atau rintangan atau bahkan bisa mengulang jalur yang sama jika tidak diimplementasikan dengan pengecekan simpul yang telah dikunjungi yang membuatnya tidak terlalu optimal.

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

1. Algoritma dimulai dengan menentukan kata awal dan kata tujuan. Validasi dilakukan untuk memastikan bahwa kedua kata memiliki panjang yang sama, jika ya, pencarian dihentikan dengan segera karena solusi sudah ditemukan. Jika tidak, algoritma menyiapkan struktur data seperti priority queue untuk menyimpan node yang perlu dieksplorasi dan hashmap untuk kata-kata yang sudah dikunjungi dan heuristik untuk mencapainya. Kata awal ditetapkan sebagai simpul awal dengan hamming distance yang dihitung terhadap kata tujuan. Proses dimulai dengan pengecekan apakah kata awal sama dengan kata akhir..
2. Sebuah priority queue digunakan untuk mengatur simpul berdasarkan nilai heuristiknya, yaitu hamming distance. Simpul dengan hamming distance

terkecil, yang menunjukkan kesamaan atau kedekatan terbesar dengan kata tujuan, akan diprioritaskan dalam proses eksplorasi.

3. Selama priority queue tidak kosong, simpul dengan estimasi cost terkecil (heuristik terendah) atau $h(n)$ terkecil diambil dan dieksplorasi. Jika simpul tersebut merupakan kata tujuan, jalur dari kata awal ke kata tujuan direkonstruksi dan proses pencarian dihentikan. Jika bukan, algoritma melanjutkan dengan menghasilkan semua tetangga dari kata tersebut yang belum pernah dikunjungi.

4. Setiap tetangga yang dihasilkan—kata-kata yang bisa dicapai dengan mengubah satu huruf dari kata saat ini—dinilai berdasarkan hamming distance mereka terhadap kata tujuan. Simpul-simpul baru ini dimasukkan ke dalam priority queue untuk dieksplorasi lebih lanjut berdasarkan heuristik mereka.

5. Proses ini terus berulang dengan mengambil simpul dari priority queue, eksplorasi, dan generasi simpul anak, hingga solusi ditemukan atau priority queue menjadi kosong, yang berarti tidak ada jalur yang mungkin dari kata awal ke kata tujuan.

Dengan menggunakan pendekatan heuristik berbasis hamming distance (berdasarkan perbedaan huruf), GBFS cenderung cepat menemukan jalur ke tujuan karena secara instingtif akan tertarik ke simpul yang tampak paling menjanjikan, meskipun tidak selalu menghasilkan jalur yang paling pendek. Kelemahan dari GBFS adalah ia bisa terjebak pada jalur yang tampak menjanjikan tetapi tidak efisien yang mengakibatkan jumlah jalur yang dihasilkan lebih banyak dan tidak optimal, atau bahkan buntu jika semua opsi yang tersisa lebih buruk dari yang sudah dijelajahi. Namun, dalam kasus di mana estimasi heuristik cukup akurat dan terkait langsung dengan tujuan, seperti dalam Word Ladder dengan hamming distance, GBFS dapat menjadi sangat efektif untuk mencapai tujuan dengan cepat.

2.3 Algoritma A* Search

A Search* memadukan keunggulan dari UCS dan GBFS, mencari jalur terpendek dengan menggunakan kombinasi dari biaya yang telah dikeluarkan dan estimasi biaya ke tujuan. Seperti UCS, A* menggunakan fungsi $g(n)$ untuk melacak biaya aktual dari simpul awal ke simpul n . Seperti GBFS, A* menggunakan fungsi heuristik $h(n)$ untuk memperkirakan biaya dari simpul n ke tujuan. Fungsi $f(n) = g(n)$

+ $h(n)$ menggabungkan kedua nilai ini untuk setiap simpul, dan simpul dengan nilai $f(n)$ terkecil selalu dipilih untuk diekspansi berikutnya.

Dengan memastikan bahwa heuristiknya adalah admissible, yaitu tidak pernah *overestimate* biaya aktual ke tujuan, algoritma A* menjamin bahwa solusi yang ditemukan tidak hanya efisien tetapi juga optimal. Heuristik yang sering digunakan dalam A* termasuk jarak lurus (dalam konteks geografis) atau heuristik yang lebih kompleks dalam aplikasi seperti perencanaan jalur dalam video game atau robotika, di mana langkah-langkah bisa memiliki biaya yang sangat beragam dan konteksnya bisa berubah-ubah.

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

1. Proses dimulai dengan pengecekan apakah kata awal sama dengan kata akhir. Jika ya, pencarian dihentikan dengan segera karena solusi sudah ditemukan. Jika tidak, algoritma menyiapkan struktur data seperti priority queue untuk menyimpan node yang perlu dieksplorasi dan hashmap untuk melacak biaya terendah yang diketahui untuk mencapai setiap kata.
2. Node awal, yang mewakili kata awal, ditambahkan ke priority queue dengan biaya awal (g) nol dan heuristik (h) yang dihitung menggunakan Hamming Distance ke kata akhir. Node ini memiliki nilai f , yang merupakan total biaya $f = g + h$, yaitu penjumlahan biaya aktual dari node awal ditambah dengan heuristik ke node tujuan.
3. Selama queue tidak kosong, node dengan nilai f terkecil (yaitu, biaya terendah saat ditambahkan dengan estimasi ke tujuan) diambil dan dieksplorasi. Ini memastikan bahwa pencarian selalu mengutamakan jalur yang tampak paling menjanjikan untuk mencapai tujuan secara efisien.
4. Untuk setiap node yang dieksplorasi, algoritma menghasilkan "tetangga" atau kata-kata yang dapat dicapai dengan mengubah satu huruf pada satu waktu. Untuk setiap tetangga yang belum dikunjungi atau yang bisa dicapai dengan biaya lebih rendah dari yang sebelumnya diketahui, algoritma mengupdate biaya mereka dan menambahkan mereka ke dalam priority queue dengan nilai f baru yang dihitung kembali.
5. Biaya untuk mencapai tetangga g adalah biaya untuk mencapai node saat ini ditambah satu langkah lagi. Heuristik h dihitung lagi sebagai Hamming

Distance dari tetangga ke kata akhir. Nilai f untuk setiap tetangga baru adalah penjumlahan dari biaya ini ($g + h$).

6. Jika kata tujuan ditemukan, algoritma mengkonstruksi kembali jalur dari kata awal ke kata akhir dengan mengikuti rantai *parent* dari node tujuan kembali ke node awal. Jika queue menjadi kosong sebelum kata akhir ditemukan, ini menandakan tidak ada jalur yang mungkin.

Algoritma A^* sangat dihargai karena keefektifannya dalam memprioritaskan eksplorasi node yang tidak hanya dekat dengan titik awal tetapi juga yang tampak mendekati tujuan, menggabungkan keunggulan algoritma pencarian yang optimal seperti Dijkstra atau UCS dengan kecepatan algoritma yang menggunakan heuristik seperti Greedy Best First Search. Dengan pendekatan ini, A^* cenderung menemukan jalur yang optimal dan lebih cepat daripada metode pencarian lain yang memerlukan waktu lebih lama untuk mengeksplorasi jalur yang kurang relevan.

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.*;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Set<String> words = WordLoader.loadWords("bin/words.txt");
        NeighborGenerator neighbor = new NeighborGenerator(words);

        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*)");

        String start, end;
        do {
            System.out.print("\nEnter start word: ");
            start = scanner.nextLine().toLowerCase();
            if (!words.contains(start)) {
                System.out.println("The word '" + start + "' is not a valid English word. Please try again.");
            }
        } while (!words.contains(start));

        do {
            System.out.print("Enter end word: ");
            end = scanner.nextLine().toLowerCase();
            if (!words.contains(end)) {
                System.out.println("The word '" + end + "' is not a valid English word. Please try again.");
            } else if (start.length() != end.length()) {
                System.out.println("The length of start and end words should be the same. Please try again.");
            }
        } while (!words.contains(end) || start.length() != end.length());

        int choice;
        do {
            System.out.print("\nSelect algorithm (1 for UCS, 2 for GBFS, 3 for Astar): ");
            try {
                choice = Integer.parseInt(scanner.nextLine());
                if (choice < 1 || choice > 3) {
                    throw new NumberFormatException();
                }
            } catch (NumberFormatException e) {
                System.out.println("Invalid input. Please enter a number corresponding to the algorithm.");
                choice = 0;
            }
        }
```

```

    } while (choice < 1 || choice > 3);

    String algorithm;
    switch (choice) {
        case 1:
            algorithm = "UCS";
            break;
        case 2:
            algorithm = "GBFS";
            break;
        case 3:
            algorithm = "ASTAR";
            break;
        default:
            algorithm = "";
            break;
    }

    SearchResult result = null;
    long startTime = System.nanoTime();
    switch (algorithm) {
        case "UCS":
            result = UCS.findLadder(start, end, neighbor);
            break;
        case "GBFS":
            result = GBFS.findLadder(start, end, neighbor);
            break;
        case "ASTAR":
            result = Astar.findLadder(start, end, neighbor);
            break;
        default:
            System.out.println("Invalid algorithm choice");
            System.exit(0);
    }

    long endTime = System.nanoTime();
    long executionTime = (endTime - startTime) / 1_000_000;

    if (result.path.isEmpty()) {
        System.out.println("No path found between " + start + " and " + end);
    } else {
        List<String> ladder = result.path.stream().map(String::toUpperCase).toList();
        System.out.println("\nSelected Algorithm: " + algorithm);
        System.out.println("Path found: " + String.join(" → ", ladder));
        System.out.println("Visited nodes: " + result.visitedNodes);
        System.out.println("Total Path found: " + ladder.size());
    }

    System.out.println("Execution time: " + executionTime + " ms");
    scanner.close();
}
}

```

Di dalam file *Main.java*, terdapat class *Main* yang memiliki method *main* yang dimana akan diinisialisasi kamus bahasa inggris dalam suatu set dan akan dilakukan pemanggilan class *NeighborGenerator* untuk mencari tetangga dari kata-kata yang ada di kamus. Kemudian akan dilakukan penginputan oleh user untuk memasukkan inputan kata awal, akhir dan juga jenis algoritma yang akan dipilih. Lalu, akan ditampilkan hasil dari

jalannya algoritma yang dipilih. File Main dibuat untuk membiarkan user untuk menjalankan program menggunakan CLI.

3.2. GUI.java (Bonus)

GUI.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;
import java.util.List;
import java.util.Set;

public class GUI extends JFrame {
    private JTextField startWordField;
    private JTextField endWordField;
    private JButton findButton;
    private JPanel resultPanel;
    private JComboBox<String> algorithmSelector;
    private JLabel algoLabel;
    private JLabel timeLabel;
    private JLabel visitedLabel;
    private Set<String> words;
    NeighborGenerator neighbor;

    public GUI() {
        words = WordLoader.loadWords("bin/words.txt");
        neighbor = new NeighborGenerator(words);
        // graph = GraphBuilder.buildGraph(words);
        createGUI();
        setupCloseKeyBinding();
    }

    private void createGUI() {
        setTitle("Word Ladder Game");
        setSize(600, 500);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        getContentPane().setLayout(new BorderLayout(getContentPane(), BorderLayout.Y_AXIS));

        JPanel contentPanel = new JPanel();
        contentPanel.setLayout(new BorderLayout(contentPanel, BorderLayout.Y_AXIS));
        contentPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 0, 10));

        JPanel topPanel = new JPanel(new GridLayout(3, 2, 5, 5));
        topPanel.add(new JLabel("Start Word:"));
        startWordField = new JTextField(15);
        topPanel.add(startWordField);

        topPanel.add(new JLabel("End Word:"));
        endWordField = new JTextField(10);
        topPanel.add(endWordField);

        topPanel.add(new JLabel("Select Algorithm:"));
        algorithmSelector = new JComboBox<>(new String[]{"UCS", "GBFS", "A*"});
        topPanel.add(algorithmSelector);

        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));
        findButton = new JButton("Solve");
        findButton.addActionListener(e → executeSearch());
        buttonPanel.add(findButton);
    }
}
```

```

        JPanel statusPanel = new JPanel();
        statusPanel.setLayout(new GridLayout(1, 3, 5, 5));
        algoLabel = new JLabel("Selected Algorithm: -", JLabel.CENTER);
        timeLabel = new JLabel("Execution Time: -", JLabel.CENTER);
        visitedLabel = new JLabel("Visited Nodes: -", JLabel.CENTER);
        statusPanel.add(algoLabel);
        statusPanel.add(timeLabel);
        statusPanel.add(visitedLabel);

        contentPanel.add(topPanel);
        contentPanel.add(buttonPanel);

        resultPanel = new JPanel();
        resultPanel.setLayout(new BoxLayout(resultPanel, BoxLayout.Y_AXIS));
        JScrollPane scrollPane = new JScrollPane(resultPanel);
        scrollPane.setPreferredSize(new Dimension(580, 300));

        contentPanel.add(scrollPane);
        contentPanel.add(statusPanel);
        add(contentPanel, BorderLayout.CENTER);
        setLocationRelativeTo(null);
    }

    private void setupCloseKeyBinding() {
        String closeKey = "CLOSE";
        InputMap inputMap = getRootPane().getInputMap(JComponent.WHEN_IN_FOCUSED_WINDOW);
        inputMap.put(KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, 0), closeKey);
        ActionMap actionMap = getRootPane().getActionMap();
        actionMap.put(closeKey, new AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent e) {
                dispose();
            }
        });
    }

    private void executeSearch() {
        resultPanel.removeAll();
        String start = startWordField.getText().toLowerCase();
        String end = endWordField.getText().toLowerCase();
        String algorithm = (String) algorithmSelector.getSelectedItem();

        algoLabel.setText("Selected Algorithm: " + algorithm);

        if (!words.contains(start) || !words.contains(end) || start.length() !=
end.length()) {
            JOptionPane.showMessageDialog(this, "Invalid input english words.");
            return;
        }

        if (start.length() != end.length()) {
            JOptionPane.showMessageDialog(this, "Word lengths are not equal.");
            return;
        }

        Runtime runtime = Runtime.getRuntime();
        long initialMemory = runtime.totalMemory() - runtime.freeMemory();

        long startTime = System.nanoTime();
        SearchResult result = null;
        try {
            switch (algorithm) {
                case "UCS":

```

```

        result = UCS.findLadder(start, end, neighbor);
        break;
    case "GBFS":
        result = GBFS.findLadder(start, end, neighbor);
        break;
    case "A*":
        result = Astar.findLadder(start, end, neighbor);
        break;
    default:
        JOptionPane.showMessageDialog(this, "Invalid algorithm choice");
        return;
    }
} catch (Exception ex) {
    JOptionPane.showMessageDialog(this, "Error during search: " + ex.getMessage());
    ex.printStackTrace();
    return;
}

long endTime = System.nanoTime();
long finalMemory = runtime.totalMemory() - runtime.freeMemory();
long memoryUsed = (finalMemory - initialMemory);
long executionTime = (endTime - startTime) / 1_000_000;
timeLabel.setText("Execution Time: " + executionTime + " ms");
visitedLabel.setText("Visited Nodes: " + (result != null ? result.visitedNodes :
0));
System.out.println("Algorithm: " + algorithm + ", Memory used: " + memoryUsed + "
bytes");

if (result != null && result.path != null) {
    displayLadder(result.path);
} else {
    JOptionPane.showMessageDialog(this, "No path found.");
}
}

private void displayLadder(List<String> ladder) {
    resultPanel.removeAll();
    if (ladder == null || ladder.isEmpty()) {
        resultPanel.setLayout(new BorderLayout());
        JLabel noPathLabel = new JLabel("No path found.");
        noPathLabel.setHorizontalAlignment(SwingConstants.CENTER);
        resultPanel.add(noPathLabel, BorderLayout.CENTER);
    } else {
        resultPanel.setLayout(new BoxLayout(resultPanel, BoxLayout.Y_AXIS));
        String reference = ladder.get(0);
        char[] lastState = reference.toCharArray();

        for (int i = 0; i < ladder.size(); i++) {
            JPanel wordPanel = new JPanel();
            wordPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
            String current = ladder.get(i);

            JLabel numberLabel = new JLabel((i + 1) + ". ");
            wordPanel.add(numberLabel);

            for (int j = 0; j < current.length(); j++) {
                JLabel label = new JLabel(String.valueOf(current.charAt(j)));
                if (current.charAt(j) != lastState[j]) {
                    label.setForeground(Color.BLUE);
                    lastState[j] = current.charAt(j);
                }
                wordPanel.add(label);
            }
        }
    }
}

```

```

        resultPanel.add(wordPanel);
        if (i < ladder.size() - 1) {
            resultPanel.add(new JSeparator());
        }
    }
    resultPanel.revalidate();
    resultPanel.repaint();
}

public static void main(String[] args) {
    SwingUtilities.invokeLater(() → {
        GUI gui = new GUI();
        gui.setVisible(true);
    });
}
}

```

Dalam file GUI.java, terdapat sebuah kelas bernama GUI. Kelas ini menginisialisasi semua komponen visual dari GUI tersebut, yang dibuat dalam metode *createGUI*. Terdapat juga metode *executeSearch* yang bertugas memanggil algoritma pencarian yang digunakan. Selain itu, ada metode *displayLadder* yang berfungsi untuk memvisualisasikan jalur yang dihasilkan oleh pencarian tersebut. File ini digunakan untuk memungkinkan user untuk menjalankan program menggunakan GUI (Graphical-User-Interface).

3.3. SearchResults.java

SearchResults.java

```

import java.util.*;

public class SearchResult {
    public final List<String> path;
    public final int visitedNodes;

    public SearchResult(List<String> path, int visitedNodes) {
        this.path = path;
        this.visitedNodes = visitedNodes;
    }
}

```

Dalam file SearchResult.java, terdapat kelas *SearchResult* yang berperan sebagai pembungkus untuk hasil dari pencarian. Kelas ini memiliki dua atribut penting: *path* dan *visitedNodes*. Atribut *path* adalah *List<String>* yang menyimpan urutan kata-kata dari kata awal hingga kata akhir yang dihasilkan oleh algoritma pencarian. Atribut *visitedNodes* mencatat jumlah simpul atau node yang telah dikunjungi selama proses pencarian, memberikan ukuran efisiensi algoritma dalam mengeksplorasi ruang pencarian. Konstruktor dari kelas ini, *SearchResult(List<String> path, int visitedNodes)*, menginisialisasi objek dengan jalur yang telah ditemukan dan jumlah simpul yang telah dikunjungi, memudahkan analisis kinerja dan visualisasi hasil pencarian.

3.4. WordLoader.java

WordLoader.java

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

public class WordLoader {
    public static Set<String> loadWords(String filename) {
        Set<String> words = new HashSet<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                words.add(line.trim().toLowerCase());
            }
        } catch (IOException e) {
            System.err.println("Error reading the word file: " + e.getMessage());
        }
        return words;
    }
}
```

Dalam file WordLoader.java, terdapat sebuah kelas yang bernama WordLoader. Kelas ini memiliki satu metode yaitu *loadWords*. Metode ini berfungsi untuk membaca dan memuat semua kata dari file teks yang diberikan dan menyimpannya dalam sebuah Set<String>. Setiap kata dibaca per baris, membuang spasi yang tidak diperlukan, dan diubah ke huruf kecil sebelum ditambahkan ke set. Penggunaan struktur data Set di sini memastikan bahwa semua kata yang dimuat adalah unik dan tidak ada duplikasi. Metode *loadWords* ini sangat berguna untuk menginisialisasi dan mempersiapkan daftar kata yang akan digunakan dalam aplikasi, seperti permainan teka-teki atau pengolah kata, di mana akses cepat ke kata-kata yang valid dan eliminasi duplikat adalah penting.

3.5. UCS.java

UCS.java

```
import java.util.*;

public class UCS {
    public static SearchResult findLadder(String start, String end, NeighborGenerator generator) {
        if(start.equals(end)){
            return new SearchResult(Collections.singletonList(start), 0);
        }

        PriorityQueue<Node> priorityQueue = new PriorityQueue<>();
        Map<String, Integer> visited = new HashMap<>();
        int visitedNodes = 0;

        priorityQueue.add(new Node(start, null, 0));
        visited.put(start, 0);

        while (!priorityQueue.isEmpty()) {
```



```

        Node current = priorityQueue.poll();

        if (current.cost ≤ visited.get(current.word)) {
            visitedNodes++;

            if (current.word.equals(end)) {
                List<String> path = getPath(current);
                return new SearchResult(path, visitedNodes);
            }

            List<String> neighbors = generator.getNeighbors(current.word);
            for (String neighbor : neighbors) {
                int newCost = current.cost + 1;
                if (!visited.containsKey(neighbor) || newCost < visited.get(neighbor))
                {
                    visited.put(neighbor, newCost);
                    priorityQueue.add(new Node(neighbor, current, newCost));
                }
            }
        }

        return new SearchResult(Collections.emptyList(), visitedNodes);
    }

    private static List<String> getPath(Node endNode) {
        List<String> path = new ArrayList<>();
        Node current = endNode;
        while (current ≠ null) {
            path.add(current.word);
            current = current.parent;
        }
        Collections.reverse(path);
        return path;
    }

    static class Node implements Comparable<Node> {
        String word;
        Node parent;
        int cost;

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

        @Override
        public int compareTo(Node other) {
            return Integer.compare(this.cost, other.cost);
        }
    }
}

```

Dalam file UCS.java, terdapat sebuah kelas yang bernama UCS. Kelas ini memiliki metode utama yang disebut *findLadder*, yang berfungsi untuk mencari jalur terpendek dari satu kata ke kata lain menggunakan algoritma Uniform Cost Search (UCS). Metode ini menggunakan *PriorityQueue* untuk menyimpan dan mengurutkan node berdasarkan biaya terendah untuk mencapai node tersebut, yang memungkinkan pencarian yang efisien dalam menemukan jalur dengan biaya terendah.

Metode *findLadder* juga memanfaatkan *Map* untuk melacak kata-kata yang telah dikunjungi dan biaya minimum untuk mencapai kata tersebut. Ini menghindari pengulangan dan optimasi jalur. Saat kata tujuan ditemukan, metode *getPath* dipanggil untuk membangun jalur terbalik dari node akhir ke node awal dengan mengikuti pointer ke node induk, yang kemudian dibalikkan untuk mendapatkan urutan yang benar.

Selain itu, kelas UCS mendefinisikan kelas internal *Node* yang mengimplementasikan antarmuka *Comparable<Node>* untuk menentukan urutan dalam *PriorityQueue* berdasarkan biaya. Setiap *Node* menyimpan kata yang diwakilinya, node induknya dalam jalur, dan biaya total dari node awal hingga node tersebut.

3.6. GBFS.java

GBFS.java

```
import java.util.*;

public class GBFS {
    public static SearchResult findLadder(String start, String end, NeighborGenerator
generator) {
        if(start.equals(end)){
            return new SearchResult(Collections.singletonList(start), 0);
        }

        PriorityQueue<Node> prioQueue = new PriorityQueue<>();
        Map<String, Boolean> visited = new HashMap<>();
        int visitedNodes = 0;

        prioQueue.add(new Node(start, null, calculateHammingDistance(start, end)));
        visited.put(start, true);

        while (!prioQueue.isEmpty()) {
            Node current = prioQueue.poll();
            visitedNodes++;

            if (current.word.equals(end)) {
                List<String> path = getPath(current);
                return new SearchResult(path, visitedNodes);
            }

            List<String> neighbors = generator.getNeighbors(current.word);
            for (String neighbor : neighbors) {
                if (!visited.containsKey(neighbor)) {
                    visited.put(neighbor, true);
                    prioQueue.add(new Node(neighbor, current,
calculateHammingDistance(neighbor, end)));
                }
            }
        }
        return new SearchResult(Collections.emptyList(), visitedNodes);
    }

    private static List<String> getPath(Node endNode) {
        List<String> path = new ArrayList<>();
        for (Node node = endNode; node != null; node = node.parent) {
            path.add(node.word);
        }
    }
}
```

```

    }
    Collections.reverse(path);
    return path;
}

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

static class Node implements Comparable<Node> {
    String word;
    Node parent;
    int heuristic;

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

    @Override
    public int compareTo(Node other) {
        return Integer.compare(this.heuristic, other.heuristic);
    }
}
}

```

Dalam file GBFS.java, terdapat sebuah kelas yang bernama GBFS, yang merupakan singkatan dari Greedy Best-First Search. Kelas ini menyediakan metode utama yang disebut *findLadder*, yang dirancang untuk mencari jalur dari satu kata ke kata lain dengan meminimalkan jumlah perubahan huruf berdasarkan heuristik tertentu. Metode ini menggunakan algoritma Greedy Best-First Search, di mana setiap langkah mencoba mendekati kata tujuan dengan memilih kata berikutnya yang memiliki *Hamming distance* terkecil dibandingkan dengan kata tujuan.

Metode *findLadder* memulai dengan memeriksa apakah kata awal sudah sama dengan kata tujuan, dan jika ya, ia langsung mengembalikan hasil dengan kata awal sebagai satu-satunya langkah dan jumlah node yang dikunjungi adalah nol. Jika tidak, ia menginisialisasi *PriorityQueue* yang menyimpan node, dengan masing-masing node dievaluasi berdasarkan heuristik *Hamming distance* dari kata itu ke kata tujuan. Heuristik ini digunakan sebagai dasar untuk membandingkan node dalam antrian prioritas, sehingga node dengan jarak terdekat ke kata tujuan selalu diproses terlebih dahulu.

Metode ini juga menggunakan *Map* untuk melacak kata-kata yang telah dikunjungi untuk menghindari pengulangan dan untuk memastikan efisiensi pencarian. Setiap kali kata tujuan ditemukan melalui proses traversal, metode *getPath* dipanggil untuk membangun jalur

dari node akhir kembali ke node awal dengan mengikuti pointer ke node induk setiap node, dan jalur ini kemudian dibalikkan untuk mendapatkan urutan kata yang benar dari awal ke tujuan.

Kelas Node yang terdapat di dalam GBFS adalah kelas internal yang menyimpan kata, node induk, dan nilai heuristik yang dihitung berdasarkan jumlah perbedaan huruf dari antar kata. Node ini mengimplementasikan antarmuka Comparable<Node> untuk memungkinkan perbandingan berdasarkan nilai heuristiknya, sehingga PriorityQueue dapat mengurutkannya dengan benar.

3.7. Astar.java

Astar.java

```
import java.util.*;

public class Astar {
    public static SearchResult findLadder(String start, String end, NeighborGenerator
generator) {
        if(start.equals(end)){
            return new SearchResult(Collections.singletonList(start), 0);
        }

        PriorityQueue<Node> priorityQueue = new PriorityQueue<>();
        Map<String, Integer> costSoFar = new HashMap<>();
        int visitedNodes = 0;
        priorityQueue.add(new Node(start, null, 0, calculateHammingDistance(start, end)));
        costSoFar.put(start,0);

        while (!priorityQueue.isEmpty()) {
            Node current = priorityQueue.poll();

            if (current.g > costSoFar.get(current.word)) {
                continue;
            }

            visitedNodes++;
            if (current.word.equals(end)) {
                List<String> path = getPath(current);
                return new SearchResult(path, visitedNodes);
            }

            List<String> neighbors = generator.getNeighbors(current.word);
            for (String neighbor : neighbors) {
                int newCost = costSoFar.get(current.word) + 1;
                if (!costSoFar.containsKey(neighbor) || newCost < costSoFar.get(neighbor))
                {
                    costSoFar.put(neighbor, newCost);
                    int priority = newCost + calculateHammingDistance(neighbor, end);
                    priorityQueue.add(new Node(neighbor, current, newCost, priority));
                }
            }
        }
        return new SearchResult(Collections.emptyList(), visitedNodes);
    }

    private static List<String> getPath(Node endNode) {
```

```

        List<String> path = new ArrayList<>();
        for (Node node = endNode; node != null; node = node.parent) {
            path.add(node.word);
        }
        Collections.reverse(path);
        return path;
    }

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

    static class Node implements Comparable<Node> {
        String word;
        Node parent;
        int g; // Cost to reach this node from the start node
        int f; // total cost = g + h(dari calculatehammingdistance)

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

        @Override
        public int compareTo(Node other) {
            return Integer.compare(this.f, other.f);
        }
    }
}

```

Dalam file *Astar.java*, terdapat sebuah kelas yang bernama *Astar*, yang mengimplementasikan algoritma A* (A-star) untuk mencari jalur yang menghubungkan dua kata. Metode utama dalam kelas ini adalah *findLadder*, yang menggunakan kombinasi biaya aktual dari kata awal ke kata saat ini (g) dan estimasi biaya dari kata saat ini ke kata tujuan (h) untuk menentukan kata berikutnya dalam jalur. Fungsi ini membuat A* efektif dalam mencari jalur terpendek dengan mempertimbangkan kedua aspek biaya tersebut.

Metode *findLadder* memulai dengan memeriksa apakah kata awal sudah sama dengan kata tujuan, dan jika ya, mengembalikan hasil langsung dengan kata tersebut sebagai jalur tunggal dan jumlah node yang dikunjungi adalah nol. Jika tidak, metode ini menginisialisasi *PriorityQueue* yang menyimpan node dengan prioritas berdasarkan nilai f, yang adalah jumlah dari g dan h (di mana h dihitung menggunakan jarak Hamming antara kata dan kata tujuan).

Selain itu, metode ini menggunakan *Map* untuk melacak biaya terendah yang ditemukan sejauh ini untuk setiap kata yang telah dikunjungi, yang memungkinkan algoritma

untuk menghindari mengunjungi kembali node dengan biaya yang lebih tinggi. Setiap kali kata tujuan tercapai, metode *getPath* dipanggil untuk membangun jalur dari node akhir kembali ke node awal dengan mengikuti pointer ke node induk dan jalur ini kemudian dibalikkan untuk mendapatkan urutan kata yang benar dari awal ke tujuan.

Kelas *Node* yang terdapat di dalam *Astar* adalah kelas internal yang menyimpan kata, node induk, biaya untuk mencapai kata itu dari kata awal (g), dan total biaya estimasi untuk mencapai kata tujuan dari kata awal (f). *Node* ini mengimplementasikan antarmuka *Comparable<Node>* untuk memungkinkan perbandingan berdasarkan nilai f, sehingga *PriorityQueue* dapat mengurutkannya secara efisien.

3.8. NeighborGenerator.java

NeighborGenerator.java

```
import java.util.*;

public class NeighborGenerator {
    private Set<String> dictionary;

    public NeighborGenerator(Set<String> 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 originalChar = chars[i];
            for (char c = 'a'; c ≤ 'z'; c++) {
                if (c ≠ originalChar) {
                    chars[i] = c;
                    String newWord = new String(chars);
                    if (dictionary.contains(newWord)) {
                        neighbors.add(newWord);
                    }
                }
            }
            chars[i] = originalChar;
        }
        return neighbors;
    }
}
```

Dalam file *NeighborGenerator.java*, terdapat sebuah kelas yang bernama *NeighborGenerator*. Kelas ini berfungsi untuk menghasilkan tetangga dari kata yang dicari, yaitu kata-kata yang dapat dihasilkan dengan mengubah satu huruf pada suatu kata. Konstruktor kelas ini menerima sebuah *Set<String>* yang berisi kamus kata-kata yang valid dan menyimpannya dalam *dictionary*.

Metode utama dalam kelas ini adalah *getNeighbors*, yang mengambil sebuah kata dan mengembalikan daftar semua kemungkinan kata yang valid berdasarkan kamus dan yang

hanya berbeda satu huruf dari kata masukan. Cara kerjanya adalah dengan mengubah setiap karakter dalam kata, satu per satu, dengan setiap huruf dari 'a' hingga 'z', kecuali huruf yang saat ini sedang diperiksa. Jika hasil perubahan adalah kata yang terdapat dalam kamus, kata tersebut ditambahkan ke daftar tetangga.

Proses ini dilakukan secara efisien dengan memanfaatkan array karakter dari kata asli, yang memungkinkan modifikasi yang mudah dan pengembalian cepat ke keadaan semula setelah setiap iterasi.

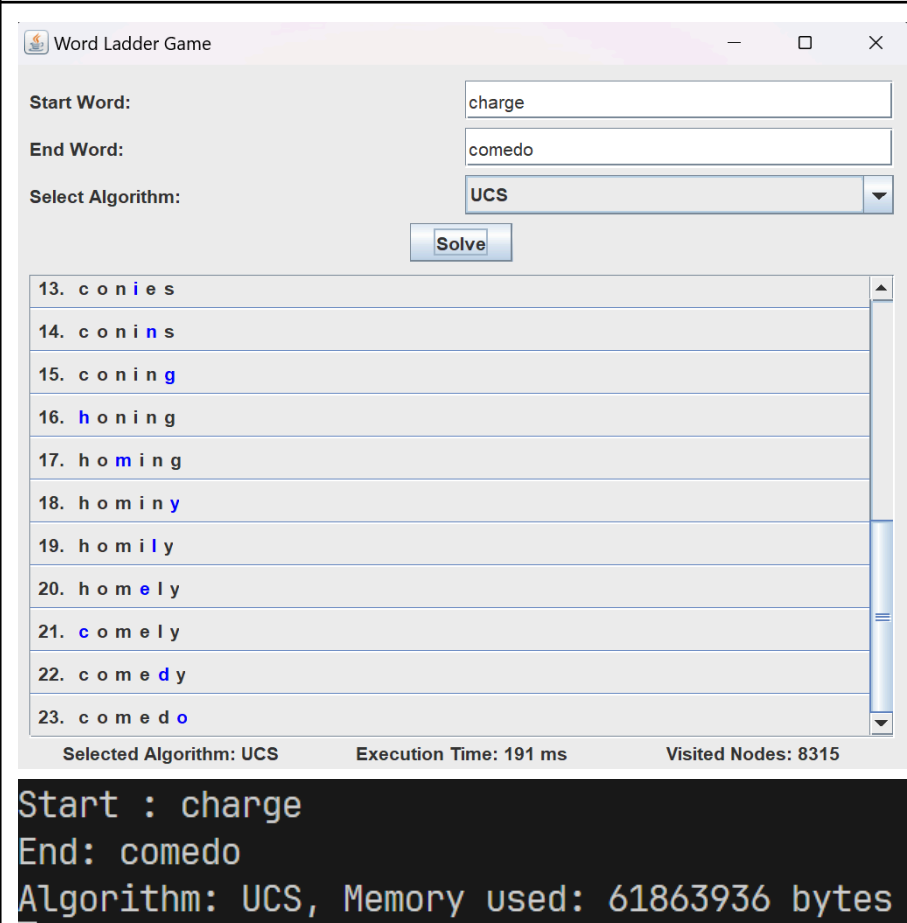
BAB IV

ANALISIS DAN PENGUJIAN

4.1. Pengujian 1

4.1.1 Pengujian Algoritma Uniform Cost Search

Tabel 1. Pengujian Algoritma Uniform Cost Search

No.	Hasil Pengujian
1	 <p>The screenshot shows a window titled "Word Ladder Game". It has input fields for "Start Word:" (charge), "End Word:" (comedo), and a "Select Algorithm:" dropdown menu set to "UCS". A "Solve" button is present. Below the button is a list of 10 words in a ladder format, numbered 13 to 23. The words are: 13. c o n i e s, 14. c o n i n s, 15. c o n i n g, 16. h o n i n g, 17. h o m i n g, 18. h o m i n y, 19. h o m i l y, 20. h o m e l y, 21. c o m e l y, 22. c o m e d y, 23. c o m e d o. At the bottom of the window, it says "Selected Algorithm: UCS", "Execution Time: 191 ms", and "Visited Nodes: 8315". Below the window, a black box contains the text: "Start : charge", "End: comedo", and "Algorithm: UCS, Memory used: 61863936 bytes".</p>

2

Word Ladder Game

Start Word:

light

End Word:

heavy

Select Algorithm:

UCS

Solve

2. s i g h t

3. s i g h s

4. s i n h s

5. s i n e s

6. l i n e s

7. l e n e s

8. l e n d s

9. l e a d s

10. l e a d y

11. l e a v y

12. h e a v y

Selected Algorithm: UCS

Execution Time: 33 ms

Visited Nodes: 4757

Start : light

End: heavy

Algorithm: UCS, Memory used: 16780936 bytes

3

Word Ladder Game

Start Word:

stone

End Word:

money

Select Algorithm:

UCS

Solve

1. s t o n e

2. s h o n e

3. p h o n e

4. p h o n y

5. p e o n y

6. p e n n y

7. b e n n y

8. b o n n y

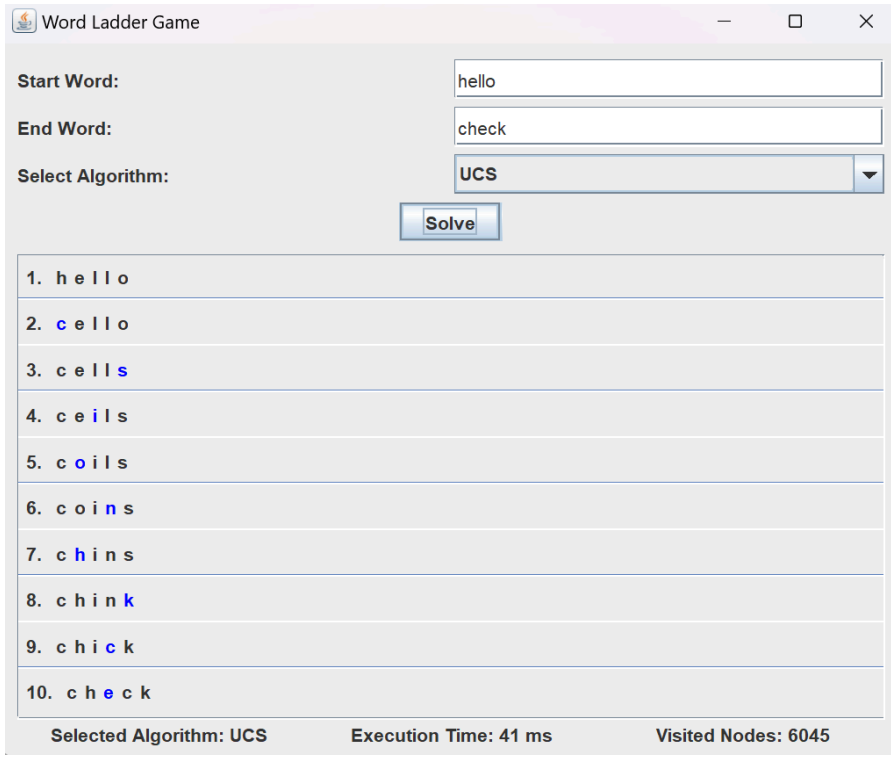
9. b o n e y

10. m o n e y

Selected Algorithm: UCS

Execution Time: 31 ms

Visited Nodes: 5045

	<pre> Start : stone End: money Algorithm: UCS, Memory used: 31982856 bytes </pre>
4	 <pre> Start : hello End: check Algorithm: UCS, Memory used: 37748720 bytes </pre>

5

Word Ladder Game

Start Word:

End Word:

Select Algorithm: UCS

1. h e l l
2. h e r l
3. h e r m
4. d e r m
5. d o r m
6. n o r m

Selected Algorithm: UCS Execution Time: 33 ms Visited Nodes: 2651

Start : hell
End: norm
Algorithm: UCS, Memory used: 422424 bytes

6

Word Ladder Game

Start Word:

End Word:

Select Algorithm: UCS

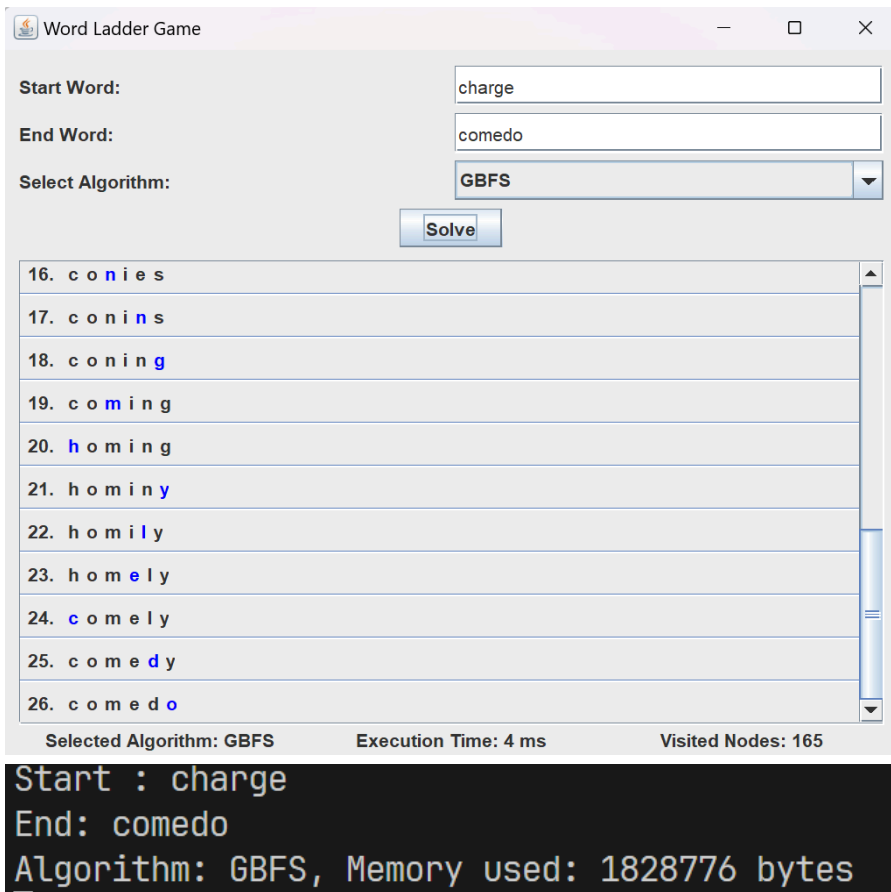
1. c r o w n
2. c r o w s
3. c r o p s
4. c o o p s
5. c o r p s
6. c o r e s
7. c o r e d
8. c o w e d
9. j o w e d
10. j e w e d
11. j e w e l

Selected Algorithm: UCS Execution Time: 37 ms Visited Nodes: 6318

	<pre> Start : crown End: jewel Algorithm: UCS, Memory used: 15863608 bytes </pre>
--	---

4.1.2 Pengujian Algoritma Greedy Best First Search

Tabel 2. Pengujian Algoritma Greedy Best First Search

No.	Hasil Pengujian
1	 <p>The screenshot shows the 'Word Ladder Game' interface. The 'Start Word' is 'charge' and the 'End Word' is 'comedo'. The 'Select Algorithm' dropdown is set to 'GBFS'. A 'Solve' button is visible. Below the button, a list of 11 words is shown, each with a number and a path of letters: 16. c o n i e s, 17. c o n i n s, 18. c o n i n g, 19. c o m i n g, 20. h o m i n g, 21. h o m i n y, 22. h o m i l y, 23. h o m e l y, 24. c o m e l y, 25. c o m e d y, 26. c o m e d o. At the bottom, it says 'Selected Algorithm: GBFS', 'Execution Time: 4 ms', and 'Visited Nodes: 165'. Below the screenshot, a black box contains the following text:</p> <pre> Start : charge End: comedo Algorithm: GBFS, Memory used: 1828776 bytes </pre>

2

Word Ladder Game

Start Word:

End Word:

Select Algorithm: GBFS

15.	h o n k y
16.	h a n k y
17.	h a n d y
18.	h a r d y
19.	h a r r y
20.	h e r r y
21.	t e r r y
22.	t e a r y
23.	l e a r y
24.	l e a v y
25.	h e a v y

Selected Algorithm: GBFS Execution Time: 0 ms Visited Nodes: 64

Start : light
 End: heavy
 Algorithm: GBFS, Memory used: 617672 bytes

3

Word Ladder Game

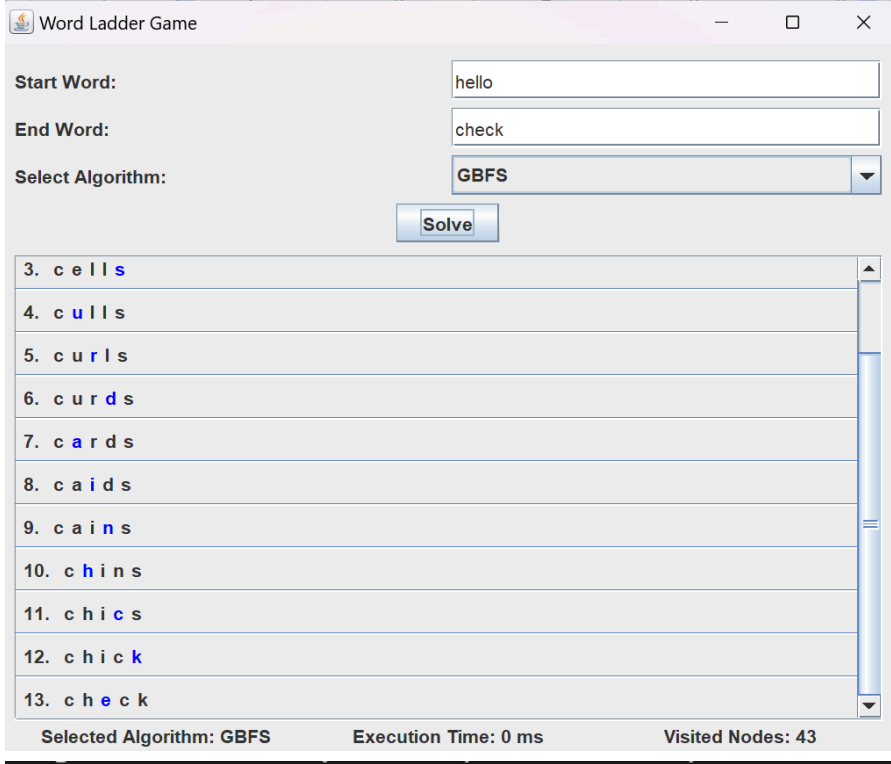
Start Word:

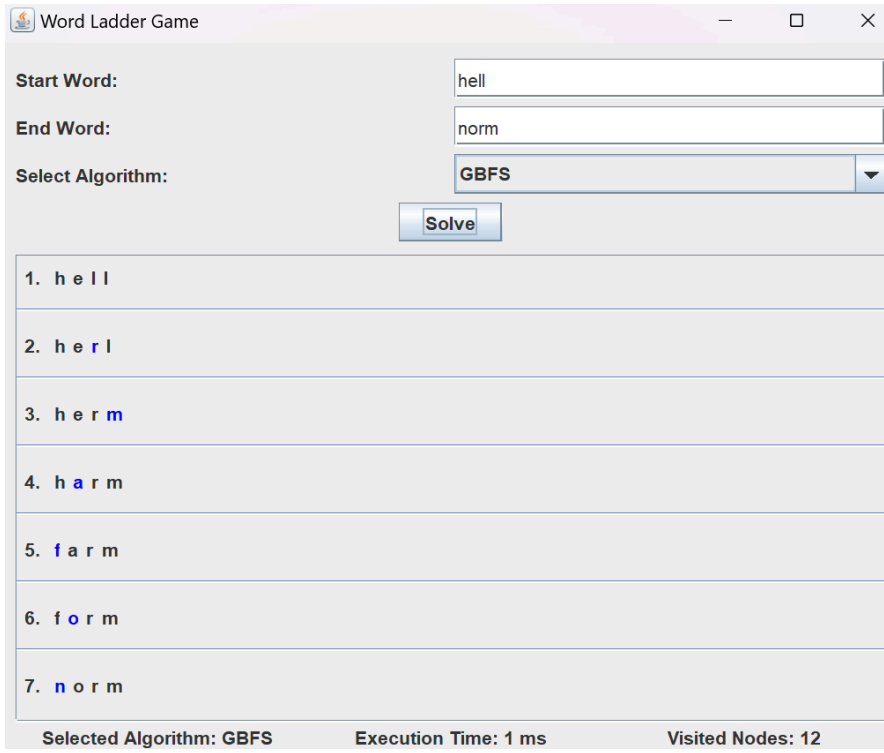
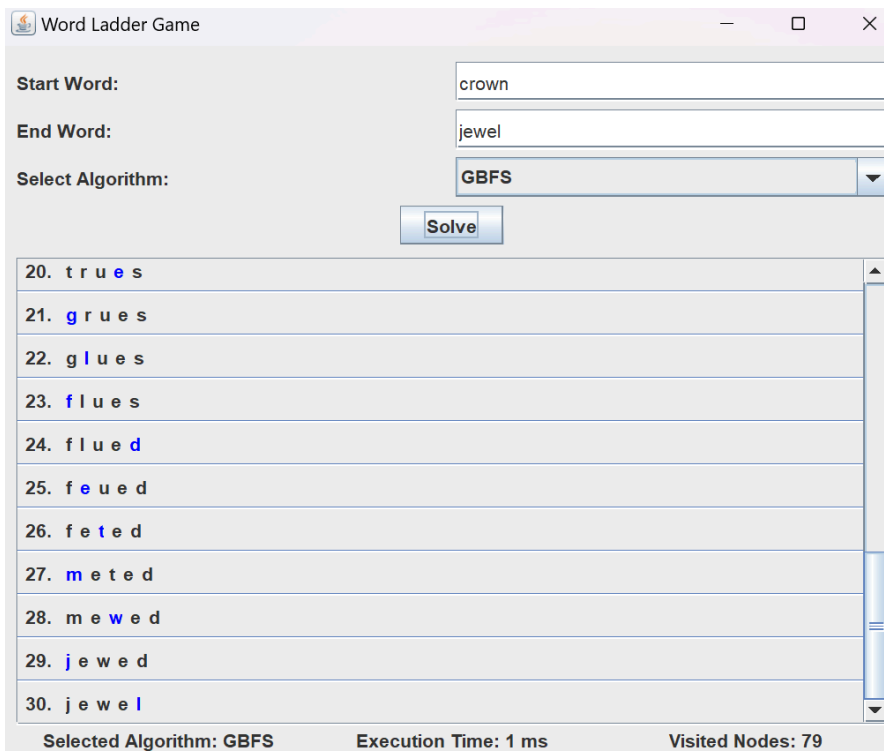
End Word:

Select Algorithm: GBFS

14.	s p i l l
15.	s p i e l
16.	s p i e r
17.	s l i e r
18.	s l y e r
19.	f l y e r
20.	f o y e r
21.	t o y e r
22.	t o n e r
23.	t o n e y
24.	m o n e y

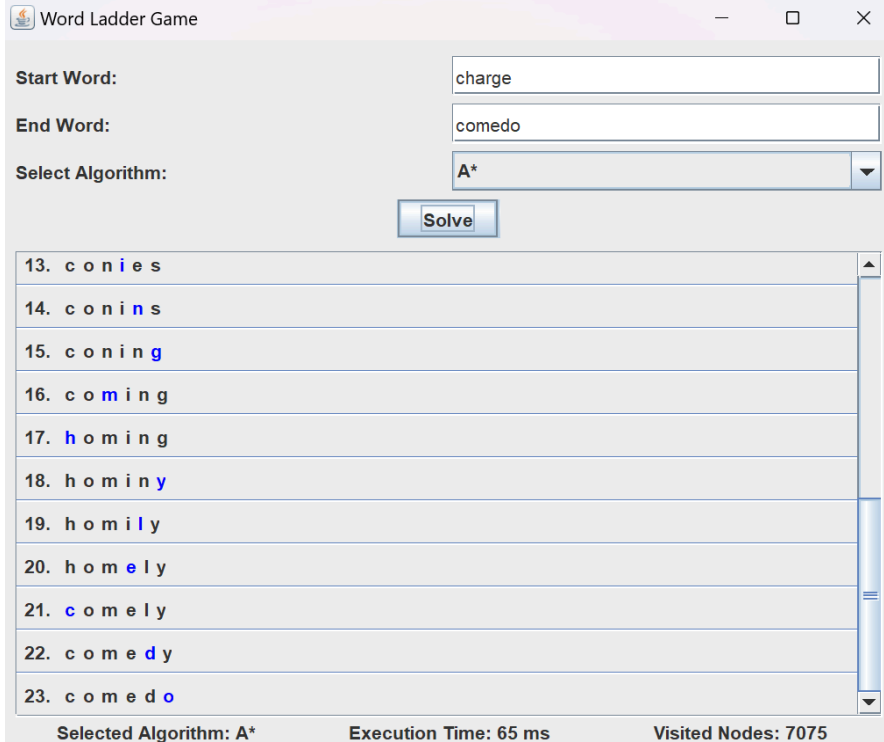
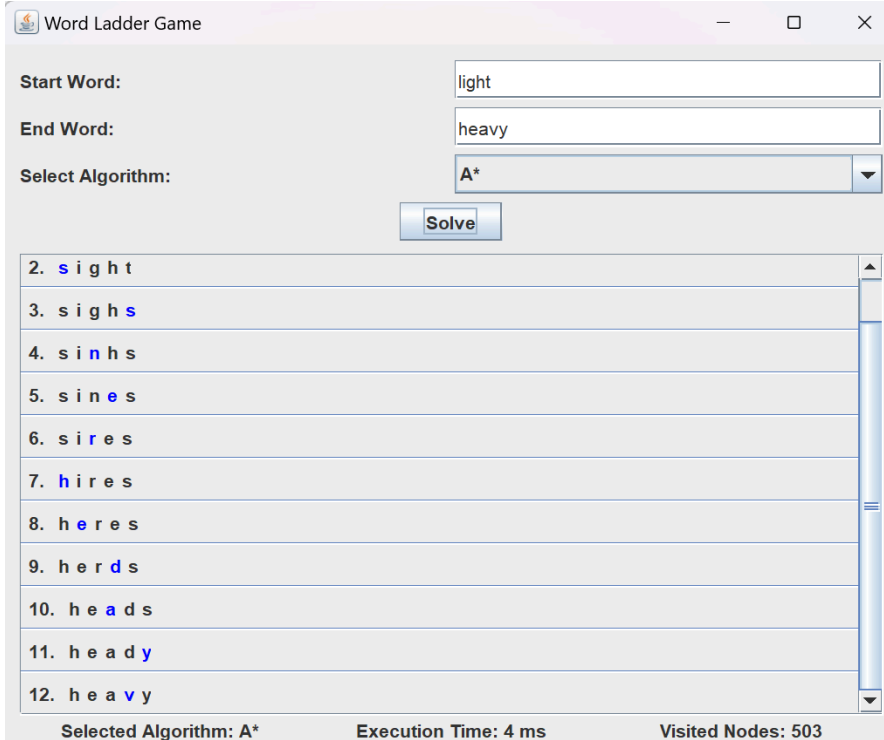
Selected Algorithm: GBFS Execution Time: 1 ms Visited Nodes: 143

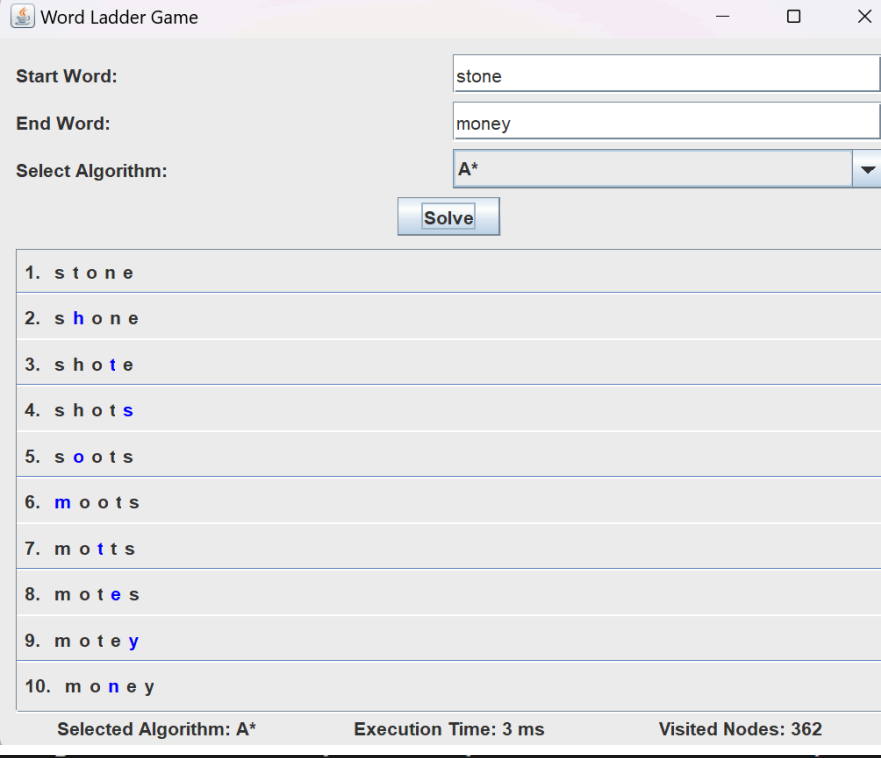
	<pre> Start : stone End: money Algorithm: GBFS, Memory used: 1047056 bytes </pre>
4	 <pre> Start : hello End: check Algorithm: GBFS, Memory used: 523528 bytes </pre>

5	 <p>Word Ladder Game</p> <p>Start Word: hell</p> <p>End Word: norm</p> <p>Select Algorithm: GBFS</p> <p>Solve</p> <ol style="list-style-type: none"> 1. h e l l 2. h e r l 3. h e r m 4. h a r m 5. f a r m 6. f o r m 7. n o r m <p>Selected Algorithm: GBFS Execution Time: 1 ms Visited Nodes: 12</p> <p>Start : hell End: norm Algorithm: GBFS, Memory used: 0 bytes</p>
6	 <p>Word Ladder Game</p> <p>Start Word: crown</p> <p>End Word: jewel</p> <p>Select Algorithm: GBFS</p> <p>Solve</p> <ol style="list-style-type: none"> 20. t r u e s 21. g r u e s 22. g l u e s 23. f l u e s 24. f l u e d 25. f e u e d 26. f e t e d 27. m e t e d 28. m e w e d 29. j e w e d 30. j e w e l <p>Selected Algorithm: GBFS Execution Time: 1 ms Visited Nodes: 79</p>

4.1.3 Pengujian Algoritma A* Search

Tabel 3. Pengujian Algoritma A*

No.	Hasil Pengujian
1	 <p>Start : charge End: comedo Algorithm: A*, Memory used: 52162752 bytes</p>
2	

	<pre> Start : light End: heavy Algorithm: A*, Memory used: 3476736 bytes </pre>
3	 <p>The screenshot shows a window titled "Word Ladder Game". It has input fields for "Start Word:" (stone), "End Word:" (money), and a "Select Algorithm:" dropdown menu set to "A*". A "Solve" button is present. Below these is a list of 10 words in the ladder, each with one letter highlighted in blue: 1. s t o n e, 2. s h o n e, 3. s h o t e, 4. s h o t s, 5. s o o t s, 6. m o o t s, 7. m o t t s, 8. m o t e s, 9. m o t e y, 10. m o n e y. At the bottom of the window, it displays "Selected Algorithm: A*", "Execution Time: 3 ms", and "Visited Nodes: 362".</p> <pre> Start : stone End: money Algorithm: A*, Memory used: 2096968 bytes </pre>

4

Word Ladder Game

Start Word:

End Word:

Select Algorithm:

1. h e l l o
2. c e l l o
3. c e l l s
4. c e i l s
5. c o i l s
6. c o i n s
7. c h i n s
8. c h i c s
9. c h i c k
10. c h e c k

Selected Algorithm: A* Execution Time: 5 ms Visited Nodes: 536

Start : hello
End: check
Algorithm: A*, Memory used: 3284752 bytes

5

Word Ladder Game

Start Word:

End Word:

Select Algorithm:

1. h e l l
2. h e r l
3. h e r m
4. d e r m
5. d o r m
6. n o r m

Selected Algorithm: A* Execution Time: 0 ms Visited Nodes: 24

Start : hell
 End: norm
 Algorithm: A*, Memory used: 0 bytes

6

Word Ladder Game
— □ ×

Start Word:

End Word:

Select Algorithm:

A*

1. c r o w n

▲

2. c r o w s

3. c r o p s

4. c o o p s

5. c o r p s

6. c o r e s

7. c o r e d

8. c o w e d

9. j o w e d

10. j e w e d

11. j e w e l

▼

Selected Algorithm: A*
Execution Time: 6 ms
Visited Nodes: 752

Start : crown
 End: jewel
 Algorithm: A*, Memory used: 5413368 bytes

4.2. Analisis

Dari hasil eksperimen, beberapa data diperoleh dari pengujian yang dilakukan. Data hasilnya disajikan dalam bentuk tabel dan divisualisasikan dalam gambar seperti berikut.

Jenis Algoritma	Nomor Percobaan	Jumlah Path	Waktu Eksekusi (ms)	Banyak node yang dikunjungi
<i>Uniform Cost Search (UCS)</i>	Percobaan 1	23	191	8315
	Percobaan 2	12	33	4757
	Percobaan 3	10	31	5045
	Percobaan 4	10	41	6045

	Percobaan 5	6	33	2651
	Percobaan 6	11	37	6318
<i>Greedy Best First Search (GBFS)</i>	Percobaan 1	26	4	165
	Percobaan 2	25	0	64
	Percobaan 3	24	1	143
	Percobaan 4	13	0	43
	Percobaan 5	7	1	12
	Percobaan 6	30	1	79
<i>A-Star (A*)</i>	Percobaan 1	23	65	7075
	Percobaan 2	12	4	503
	Percobaan 3	10	3	362
	Percobaan 4	10	5	536
	Percobaan 5	6	0	24
	Percobaan 6	11	6	752

Dari data yang diberikan, terlihat bahwa terdapat perbedaan signifikan dalam performa tiga algoritma yang digunakan untuk memecahkan permainan Word Ladder, yaitu Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A-Star (A*). Dalam analisis ini, akan dijelaskan lebih detail mengenai hasil dari masing-masing algoritma berdasarkan jumlah path yang berhasil ditemukan, waktu eksekusi, dan banyaknya node yang dikunjungi.

Uniform Cost Search (UCS) menunjukkan kecenderungan untuk menghasilkan solusi yang relatif optimal yang dimana optimalitas diukur dari jumlah path yang dihasilkan lebih sedikit dibandingkan dengan algoritma GBFS. Namun, UCS memerlukan waktu yang lebih lama, yang menjadikannya algoritma dengan waktu eksekusi paling lambat di antara ketiga algoritma ini. Hal ini terlihat dari data di mana UCS memiliki jumlah path yang paling sedikit dalam sebagian besar percobaan (misalnya hanya 6 path pada percobaan 5). Namun, algoritma ini juga memiliki waktu eksekusi yang paling lama dan jumlah node yang paling banyak dikunjungi yang menunjukkan bahwa UCS membutuhkan lebih banyak sumber daya komputasi karena menjelajahi semua kemungkinan jalur secara eksplisit tanpa preferensi arah tertentu, sehingga menjadikannya sangat menyeluruh tetapi tidak efisien waktu.

Greedy Best First Search (GBFS) sebagaimana dilihat dari hasil pengujian adalah yang paling cepat dalam hal waktu eksekusi, yang dapat dilihat dari waktu eksekusi yang sangat rendah di semua percobaannya karena jumlah node yang dikunjungi lebih sedikit. Misalnya, hanya membutuhkan 12 ms pada percobaan 5. Namun, metode ini juga menunjukkan jumlah path yang lebih banyak dan tidak konsisten, serta mengunjungi node yang lebih sedikit, yang mengindikasikan pendekatan yang lebih heuristik dan kurang sistematis. GBFS cenderung memilih jalur yang secara langsung tampak mengarah ke tujuan berdasarkan heuristiknya, tanpa mempertimbangkan keseluruhan jalur dan node lain yang mungkin memberikan solusi yang lebih optimal, seringkali menghasilkan solusi yang tidak optimal.

A-Star (A^*) terlihat sebagai algoritma yang paling optimal jika dilihat dari kombinasi antara jumlah path, waktu eksekusi, dan jumlah node yang dikunjungi. Algoritma ini secara konsisten menemukan jalur dengan jumlah path yang rendah sama seperti algoritma UCS dan waktu eksekusi yang cukup cepat pula (misalnya hanya 3 ms pada percobaan 3). Selain itu, jumlah node yang dikunjungi oleh A^* juga lebih rendah dibandingkan UCS namun lebih tinggi dari GBFS, menunjukkan bahwa A^* berhasil menyeimbangkan antara kedalaman dan kecepatan pencarian. Ini menunjukkan bahwa heuristik yang digunakan pada A^* admissible. Algoritma ini tidak hanya menghasilkan jalur yang lebih pendek tapi juga memiliki waktu eksekusi yang relatif cepat, menempatkannya di posisi yang ideal sebagai algoritma yang paling efektif untuk permainan Word Ladder. Ini karena A^* menggabungkan keunggulan dari UCS dan GBFS, menggunakan heuristik yang membantu dalam memandu pencarian, sehingga bisa efisien tanpa mengorbankan keakuratan pencarian solusi.

Penyebab perbedaan utamanya terletak pada cara algoritma mengevaluasi dan memilih node selanjutnya untuk dijelajahi. UCS tidak menggunakan heuristik dan mengeksplorasi semua node yang memungkinkan, mirip dengan bfs yang melakukan pencarian secara menyeluruh. Ini membuatnya sangat akurat tetapi lambat, memerlukan waktu eksekusi paling lama di antara yang lainnya. GBFS menggunakan heuristik yang sangat bergantung pada estimasi awal, membuatnya lebih cepat daripada kedua algoritma lainnya, tetapi hasilnya kurang optimal dan kurang akurat karena hanya mempertimbangkan biaya estimasi heuristik. Sementara itu, A^* menggabungkan kedua pendekatan tersebut dengan menggunakan heuristik untuk memperkirakan biaya dari node saat ini ke tujuan, sehingga memberikan keseimbangan yang baik antara kecepatan dan ketepatan yang membuatnya paling optimal di antara UCS dan GBFS.

Dalam segi memory yang digunakan, dari pengujian yang dilakukan, terlihat bahwa UCS memerlukan memori paling banyak, diikuti oleh A* dan GBFS. Hal ini berkorelasi dengan jumlah node yang dikunjungi oleh masing-masing algoritma selama proses pencarian solusi.

Pada algoritma UCS, konsumsi memori yang tinggi dapat dijelaskan oleh mekanisme pembangkitan node yang berkesinambungan dan bertingkat secara depth. Sifat algoritma UCS ini mirip dengan Breadth-First Search (BFS), yang mana dilakukan ekspansi node secara melebar. Karena setiap node pada setiap tingkat harus dipertimbangkan tanpa memprioritaskan mana yang lebih mendekati solusi, UCS mengalami ekspansi memori secara eksponensial. Ini mengakibatkan ruang kompleksitas yang besar karena setiap node yang dibangkitkan disimpan dalam memori.

Sedangkan Algoritma GBFS, mengutamakan pencarian berdasarkan heuristic yang hanya mengejar node yang paling menjanjikan tanpa mempertimbangkan kedalaman secara keseluruhan. Ini membuat GBFS beroperasi lebih seperti Depth-First Search (DFS) dalam hal penggunaan memori, karena hanya mengembangkan path yang secara heuristic tampak paling dekat dengan solusi tanpa mengembangkan semua kemungkinan secara merata. Oleh karena itu, GBFS memerlukan ruang kompleksitas yang lebih kecil dibandingkan dengan UCS dan A*.

Di sisi lain, algoritma A* mengurangi beban memori dengan memasukkan heuristic yang menilai biaya estimasi dari setiap node ke solusi. Walaupun A* juga melakukan pembangkitan node secara bertingkat, ada seleksi yang lebih ketat berdasarkan estimasi biaya, sehingga tidak semua node dibangkitkan seperti dalam UCS. Ini menghasilkan penggunaan memori yang lebih efisien dibandingkan UCS, sambil tetap menjaga kemungkinan mencapai solusi yang optimal.

Dari analisis ini, kita dapat melihat bahwa ada korelasi positif antara jumlah node yang dikunjungi dan memori yang dibutuhkan. Algoritma yang mengunjungi lebih banyak node cenderung menggunakan lebih banyak memori, dan sifat pengembangan node dari setiap algoritma secara signifikan mempengaruhi efisiensi memori mereka.

4.3. Penjelasan Bonus

Bonus yang dikerjakan pada tugas kecil ini adalah GUI. Aplikasi GUI ini dirancang menggunakan Java bersama dengan *framework* Swing untuk komponen antarmuka grafisnya.

GUI mencakup bidang teks *JTextField* untuk pengguna memasukkan kata awal dan akhir. Menu dropdown *JComboBox* memungkinkan pengguna memilih algoritma pencarian (UCS, GBFS, A*). Tombol Solve *JButton* memulai pencarian berdasarkan masukan dan algoritma yang dipilih. Hasil ditampilkan dalam *JPanel* yang diperbarui dengan jalur path atau lintasan kata yang ditemukan, menggunakan warna untuk menyoroti perubahan setiap karakter dalam lintasan tersebut.

Aplikasi ini juga memeriksa apakah kata-kata masukan valid dan memiliki panjang yang sama sebelum melanjutkan dengan pencarian. Jika masukan tidak valid, pesan kesalahan ditampilkan menggunakan *JOptionPane*. Saat mengklik tombol, aplikasi mengukur waktu eksekusi dan penggunaan memori oleh ketiga algoritma tersebut. Tergantung pada algoritma yang dipilih, strategi pencarian yang berbeda dieksekusi untuk menemukan jalur atau lintasan terpendek antara dua kata.

Tata letak GUI diperbarui secara dinamis berdasarkan hasil pencarian. Jika jalur ditemukan, itu ditampilkan dengan setiap kata dalam urutan disorot untuk menunjukkan perbedaan dari kata sebelumnya. Aplikasi juga menangani penutupan jendela dengan pengikatan kunci ke tombol ESC, meningkatkan interaksi pengguna dengan memungkinkan cara mudah untuk keluar dari aplikasi ini.

BAB V

PENUTUP

5.1. Kesimpulan

Dari pengujian yang dilakukan terhadap tiga algoritma pencarian yaitu Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A* dalam aplikasi word ladder, terdapat beberapa temuan penting. Pertama, UCS, yang dikenal dengan kemampuannya menghasilkan solusi optimal, cenderung membutuhkan waktu dan memori yang lebih besar untuk menyelesaikan tugas. Hal ini terjadi karena UCS menjelajahi semua kemungkinan jalur secara menyeluruh pada setiap level pencarian, sehingga seringkali memerlukan sumber daya yang lebih intensif dan besar.

Kedua, GBFS menunjukkan performa yang sangat cepat dalam eksekusinya karena algoritma ini sangat bergantung pada heuristik yang berfokus pada tujuan langsung. Namun, pendekatan ini sering kali tidak menghasilkan jalur yang optimal karena mengabaikan total biaya yang telah dikeluarkan dan hanya mementingkan heuristiknya yang bisa menyebabkan pengabaian beberapa jalur yang lebih efisien dan membuatnya menjadi tidak optimal.

Ketiga, A* muncul sebagai algoritma yang paling seimbang, mengintegrasikan biaya yang telah dikeluarkan dengan estimasi biaya ke tujuan (heuristik). Hal ini memungkinkan A* untuk secara efisien menemukan jalur yang optimal dengan menggunakan sumber daya secara lebih bijak, baik dari segi memori maupun waktu, dibandingkan dengan UCS. Dalam aplikasi word ladder yang diuji, A* secara konsisten mengungguli UCS dalam hal efisiensi waktu tanpa mengorbankan akurasi atau optimalitas jalur yang dihasilkan.

Oleh karena itu, dalam memilih algoritma yang sesuai untuk aplikasi word ladder, faktor-faktor seperti kebutuhan akan kecepatan, penggunaan memori, dan keakuratan dalam menemukan solusi yang optimal harus dipertimbangkan. GBFS bisa menjadi pilihan yang efektif untuk kasus di mana kecepatan adalah aspek yang paling krusial, meskipun mungkin tidak selalu menghasilkan solusi yang paling optimal. Sementara itu, UCS cocok untuk situasi di mana diperlukan jaminan mendapatkan jalur terpendek yang mutlak, meskipun dengan pengorbanan lebih besar dalam hal sumber daya. A*, dengan pendekatannya yang holistik, menawarkan keseimbangan yang ideal di mana solusi yang diberikan optimal dan waktu yang dihasilkan cepat, dan yang terpenting tidak memakan banyak sumber daya atau memori.

5.2. Saran

Untuk memperbaiki kinerja dan efisiensi sistem pencarian word ladder, ada baiknya mengembangkan heuristik yang lebih tepat untuk algoritma GBFS dan A*, mengoptimalkan manajemen memori pada UCS, serta menggunakan cache untuk mengurangi pengulangan perhitungan. Penerapan teknik paralel juga bisa mempercepat proses dan memaksimalkan penggunaan perangkat keras yang ada. Selain itu, melakukan evaluasi dan pengujian secara terus-menerus dengan berbagai dataset adalah penting untuk menjamin sistem bisa beradaptasi dan tetap relevan dalam berbagai kondisi penggunaan.

DAFTAR REFERENSI

- Munir, Rinaldi. Penentuan rute (Route/Path Planning) - Bagian 1. Diakses 4 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 4 Mei 2024.
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>

LAMPIRAN

Link Github: https://github.com/Filbert88/Tucil3_13522021.git

Tabel Kelayakan Program

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	✓	